USENIX

Proceedings of the 15th USENIX Security Symposium

Vancouver, B.C., Canada, July 31–August 4, 2006

conference

proceedings

15th USENIX Security Symposium

Vancouver, B.C., Canada

July 31–August 4, 2006

Sponsored by

The USENIX Association

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For additional copies of these proceedings contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 • FAX: 510-548-5738 • office@usenix.org • http://www.usenix.org

The price is $35 for members and $45 for nonmembers.
Outside the U.S.A. and Canada, please add $15 per copy for postage (via air printed matter).

Thanks to Our Student Grant Program Sponsor

Microsoft® Research

Thanks to Our Media Sponsors

ACM Queue
Addison-Wesley Professional/
Prentice Hall Professional
Dr. Dobb's Journal
GRIDtoday
HPCwire
IDG World Expo
IEEE Security & Privacy
ITtoolbox
Linux Journal
No Starch Press
OSTG
SNIA
StorageNetworking.org
Sys Admin

ISBN 1-931971-45-5

USENIX Association

Proceedings of the 15th USENIX Security Symposium

July 31–August 4, 2006
Vancouver, B.C., Canada

Conference Organizers

External Reviewers

15th USENIX Security Symposium
July 31–August 4, 2006
Vancouver, B.C., Canada

Index of Authors

Message from the Program Chair

Dear colleagues,

It is my pleasure to report that this year's USENIX Security Symposium continues the tradition of excellence established in previous years.

This year we received 179 submissions (close to the record of 184 submissions, in 2004). Each paper was reviewed by at least three program committee members. Papers by program committee members (who could be authors or co-authors in at most two submissions) were reviewed by at least four other members. Often, we solicited the opinion of outside experts. At the end of March, the program committee met as a group over two days at Columbia University (in New York) to discuss the reviews and put together the conference program. In the end, only 22 papers could be accepted, representing an acceptance rate of only 12.3%.

These numbers give only one indication of the selectivity and competitiveness of the conference. More importantly, the quality of many of the papers that we could not accept was also very high; I am confident that with only a little (if any) additional work, many will be accepted and appear in other high-quality conferences. In this, I hope they will have benefited from the hard work of the "dream team" of program committee members that I had the pleasure to preside over. Everyone tried and largely succeeded in completing their reviews on time and in providing useful and constructive feedback to the authors of all papers. I wish to thank the whole program committee for the collegiality, diligence, responsiveness, and enthusiasm they exhibited throughout this gruelling, but also extremely rewarding, process. You can be proud of the results—I know I am!

However, let me bring up a more somber issue. This year, based on previous years' experiences, we decided to bring into effect a stricter and (more importantly) more explicit policy with respect to plagiarism and publication dishonesty (including multiple submissions of the same work to concurrent conferences). Furthermore, USENIX put into effect a "sanctions" process: at the recommendation of the chair and the program committee, individuals found in violation of the policy may be prohibited from submitting to future USENIX conferences for a number of years. The details of the policy are contained in the Call for Papers, although we expect future refinements as we receive feedback and gain experience (which, hopefully, will remain limited!).

Of course, USENIX Security includes more than the technical papers. Patrick McDaniel and Gary McGraw put together a very attractive and exciting invited talks track. Doug Szajda is coordinating the ever-popular and thought provoking Work-in-Progress (WiPs) reports. Radu Sion is organizing this year's poster session, back by popular demand. Richard Clarke will be giving the keynote address for the conference. I am sure you will find this year's Symposium, held in beautiful Vancouver, an exciting and rewarding event.

In every previous USENIX proceedings foreword (for Security as well as other conferences), there is a paragraph describing how great the USENIX staff is. Let me say that none of these acknowledgments, including what I am writing, do these folks any justice. They are truly amazing in all respects. Having served as both general and program chair for conferences that do not have such dedicated and hard-working staff, I am eternally grateful to Ellie Young, Casey Henderson, Peter Collinson, Jane-Ellen Long, Tara Mulligan, Anne Dickison, and Jennifer Joost for all their patience and support. If not for them, I would not have enjoyed this process as much as I did. I would also like to thank Matt Blaze (PC Chair in 2004, USENIX Board Member) for his help in dealing with various thorny issues, and for his guidance and advice to me throughout the process.

I hope to see you all in Vancouver!

Angelos Keromytis, Columbia University
Program Chair

A Usability Study and Critique of Two Password Managers

Sonia Chiasson and P.C. van Oorschot
School of Computer Science, Carleton University, Ottawa, Canada
chiasson@scs.carleton.ca

Robert Biddle
Human Oriented Technology Lab, Carleton University, Ottawa, Canada

Abstract

We present a usability study of two recent password manager proposals: PwdHash (Ross et al., 2005) and Password Multiplier (Halderman et al., 2005). Both papers considered usability issues in greater than typical detail, the former briefly reporting on a small usability study; both also provided implementations for download. Our study involving 26 users found that both proposals suffer from major usability problems. Some of these are not "simply" usability issues, but rather lead directly to security exposures. Not surprisingly, we found the most significant problems arose from users having inaccurate or incomplete mental models of the software. Our study revealed many interesting misunderstandings – for example, users reporting a task as easy even when unsuccessful at completing that task; and believing their passwords were being strengthened when in fact they had failed to engage the appropriate protection mechanism. Our findings also suggested that ordinary users would be reluctant to opt-in to using these managers: users were uncomfortable with "relinquishing control" of their passwords to a manager, did not feel that they needed the password managers, or that the managers provided greater security.

1 Introduction

Several recent password managers, intended for protecting web accounts, generate strong passwords (i.e., more resistant to dictionary and/or social engineering attacks) from weaker but more easily remembered user-chosen passwords. They can also facilitate safer re-use of passwords across accounts by using various forms of salts such as those derived from target site domain names. The expectation is that these password managers increase security. Does this expectation materialize when real users are involved? Can ordinary users actually use these systems? Do they want to? These are some of the questions which we address in this paper, as we carry out a usability study of two password managers – PwdHash [24] and Password Multiplier [11].

Despite the inadequacies of passwords from a security viewpoint, they are becoming more common. This is causing additional challenges for ordinary users who accumulate password-protected accounts for a growing number of services and web sites. This also increases security risks for several reasons. One is that passwords are commonly reused across accounts; thus a password used for a low-security site, easily compromised by an attacker, may allow access to a higher-security site. Phishing attacks on passwords have increased dramatically over the past three years, complicating matters further. On top of these issues, it is well known that security is viewed as an obstacle by many users because security is rarely a user's primary goal. When security procedures impede users' end goals, users bypass security [2, 5, 26]. Thus usability is an increasingly important aspect of security in general, and password systems in particular.

Our objective was to examine if any progress had been made in addressing usability in security, seven years after "Why Johnny Can't Encrypt" [31]. Since novice users are often at increased risk from security vulnerabilities, we sought tools aimed at novice users as a logical starting point. Two password manager projects in particular piqued our interest: PwdHash and Password Multiplier. Both were published in 2005, had a significant focus on usability and positioned it as a major objective, and had publicly available implementations. Although the implementations that we tested are still in beta versions, their authors state that these should help novice computer users protect their passwords. We decided to carry out an independent study to test the effectiveness of the two systems from a usability standpoint – could they be successfully used, and did any security problems arise due to usability problems?

We found discrepancies between the usability claims of the published password manager papers and the results of our study. We uncovered numerous usability problems with the interfaces, some of which create security exposures. Many of the usability issues found are not particular to security, and are addressed in the existing Human-Computer Interaction (HCI) and usability literature. This suggests that the existing gap between the HCI and security communities hurts the latter due to a lack of awareness (or at least application) of the HCI usability literature, and leads only to rediscovery of well-known principles. Research effort would of course be better spent focusing on those usability issues that are unique to security interfaces.

The authors of both password managers state that security interfaces should be as transparent as possible; resulting in absolutely minimal or no change in user experience. Our study reveals that designing towards this belief can not only contribute to usability problems, but can lead to "dangerous errors" [31];[1] we conclude that this idea requires closer scrutiny and validation.

OUR CONTRIBUTIONS. We add to the relatively sparse, albeit growing, set of published usability studies in the security literature. We carry out an independent usability test of two proposed password mechanisms which have received significant attention within the security community: PwdHash [24] and Password Multiplier [11]. The results of our larger study directly contradict the findings reported for PwdHash (no user study was published for Password Multiplier), and suggest that both earlier papers (1) over claim the actual usability of their mechanisms as provided in their own publicly available implementations; and (2) result in "dangerous errors" with serious security implications. Our work reiterates the necessity of actual user studies before concluding that new security mechanisms are usable. It also raises the question of what the standard should be in the security community: should usability tests become a requirement for new authentication proposals, if a major claim is usability? Aside from usability and security considerations per se, we also provide interesting results on how users felt about these mechanisms: about giving up "control" of their passwords to these mechanisms, their perceived security, and their perceived necessity.

ORGANIZATION. Section 2 reviews the two password managers we evaluate. Section 3 first briefly provides context on usability testing, and then details our study methodology. Section 4 presents the data collected from interaction with participants, our interpretation and analysis of which is provided in Section 5. Section 6 provides further discussion and recommendations. Section 7 discusses related work. Section 8 gives concluding remarks.

2 Password Managers

Passwords are the most common form of authentication on computers today [23]; yet they are far from the most secure. Cognitive demands increase as people use more computer systems, leading to poorer password choices in an attempt to manage the load. Encouraging stronger passwords through password rules or advice on selecting good passwords does not help users remember the stronger passwords. Encouraging the use of passphrases similarly does not mitigate the problem of matching multiple passwords and multiple accounts.

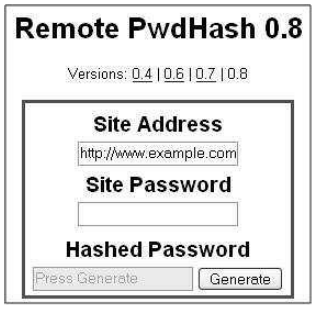


Figure 1: Remote web site for generating PwdHash passwords

One proposed solution to these password problems is *password managers*. One class of these managers maps low-entropy (easy to remember) user passwords into stronger passwords (more resistant to dictionary attacks), and may also generate site-specific passwords partially dependent on the domain name of the site (protecting against some phishing attacks). Several password managers exist in different formats: stand-alone applications (e.g., Site Password [13]), browser plug-ins (e.g., Password Maker [15]), browser scripts (e.g., Password Composer [14]), and bookmarklets (e.g., Password Generator [33]). PwdHash and Password Multiplier are two recent additions to the list. They are both browser plug-ins claiming to make it easier for users to have secure passwords without having to remember a number of complicated passwords.

2.1 PwdHash

PwdHash [24] is a browser plug-in that applies a cryptographic hash at the client machine to generate strong passwords based on the user's entered password and the site domain. The new stronger password is sent to the target site. No server-side changes are needed. Users activate the installed plug-in by adding the @@ prefix to passwords they want "protected", or by pressing the F2 key before entering their password. Once passwords are "protected", the users no longer know their effective passwords; thus PwdHash must always be used to log in to the web accounts. The authors [24] provide a web site where users can remotely generate their protected password for times when they are logging on to a site from a computer without the plug-in (see Figure 1). This generated password is copied and pasted by the users into the target password field.

Users must initially log on to each web account they want to protect, and change their password. They enter @@ in front of their new password, which activates

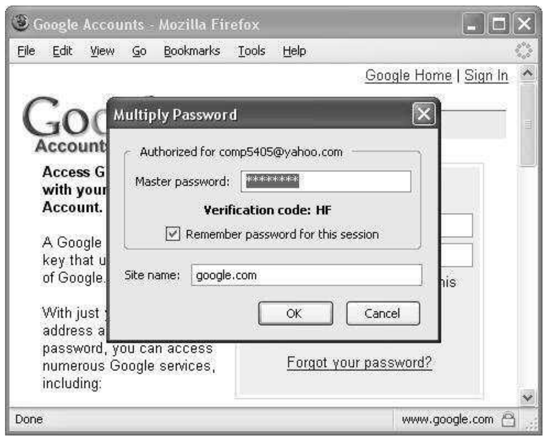
PwdHash and generates a new "strong" password. Users later changing their protected password for a given site use that web site's Change Password interface, but with @@ as a prefix to both their current and new passwords.

PwdHash is also designed to prevent certain JavaScript attacks. It processes the input stream, scanning for the @@ character, processing so-designated passwords, and replacing the input, before JavaScript would have access. PwdHash also protects against phishing attacks by using a hash salt based on the domain name of the target web site. If users enter their password at a fraudulent site, that site's domain will be used as the salt.

An implementation is publicly available. The reported user study of five users found that participants experienced little difficulty using the plug-in and that the only usability problems were observed with the remote interface. The approach was positioned as unique in that it implements password hashing within the browser, without additional window pop-ups or having a visible software interface. The PwdHash authors believe that this improves its usability: *"We can reduce the threat of password attacks with no server changes and little or no change to the user experience. Since the users who fall victim to many common attacks are technically unsophisticated, our techniques are designed to transparently provide novice users with the benefits of password practices that are otherwise only feasible for security experts"* [24].

2.2 Password Multiplier

Password Multiplier (hereafter identified as P-Multiplier) is a second password manager intended to help users generate strong passwords for web accounts [11]. It is a plug-in for Mozilla's Firefox web browser, requiring no changes to the servers; all computation is done client-side. As with PwdHash, a cryptographic hash function is applied, generating strong passwords for the user's accounts. While a primary design goal of PwdHash is to protect web site passwords against phishing attacks, P-Multiplier is intended to generate and manage strong passwords from a single user-chosen password for arbitrary password-protected applications (i.e, not restricted to web site passwords); this difference does not affect our usability study, and the plug-in we tested is restricted to web site passwords. P-Multiplier uses a master username and master password so that users need only remember one password for all of their accounts. A protected password is generated based on the master username, master password, and the target site domain name. Users activate the plug-in by double-clicking on the password field or pressing Alt+P while the cursor is in that field. This opens a small dialog box. The master username is entered automatically (it is set upon installation).



Authorized for comp5405@yahoo.com
Master password: ********
Verification code: HF
☑ Remember password for this session
Site name: google.com

Figure 2: The dialog box for P-Multiplier is activated by double-clicking on the password field

Users enter the master password (see Figure 2). When users click the OK button, the dialog box closes and the generated password is automatically placed into the web site's password field. The background colour of the web site's password field changes to signal the entry of the protected password.

Users must switch each account password over to a protected password generated by P-Multiplier, by using the Change Password interface of each web site. They double-click on the new password field to activate P-Multiplier and generate their new password. To update any password after it is protected, users must modify the "site name" argument used to generate the password (e.g., for a google.com password, change the site name in the P-Multiplier dialog box to google.com-two). Users thereafter must remember to enter their changed site name each time they log in to this particular site.

To help protect against dictionary attacks, P-Multiplier uses a two-stage process which in the first stage, massively iterates the hash function, with the intent to slow down any attack on the master password. This introduces a 100-second delay when first installing the plug-in, while it computes a first hash result which is cached on the local computer. Passwords later generated on this computer use the cached first hash, as well as the master secret again, as input to a second (less massively iterated) hash operation. Legitimate users experience the long delay only on new installations of the plug-in (not per-login). The idea is that attackers must compute the hash function from scratch each time they test a new master password, encountering the long delay with each trial.

For use from remote (secondary) computers, users must download and install the plug-in on the remote computer, and on it enter the correct master password and username in order to generate their protected passwords (and experience the 100s first-stage delay). No alternative is provided for users unable to install software (this is positioned as a relatively rare occurrence by the

P-Multiplier authors).

No usability testing was reported in publication [11] or on the P-Multiplier web site [12]. When discussing usability, they acknowledge that transportability is a necessary characteristic. They assert that PwdHash and P-Multiplier are equally transportable, however their software requires installation on a remote site (unlike PwdHash).

3 Study Methodology

To investigate the usability of these password managers, we conducted a study with participants who would be typical users of these systems. We first provide some background on usability studies, then describe our study methodology in detail.

3.1 Usability Testing

Since it is not a mainstream topic for most security researchers, we begin by briefly providing some background on usability studies. There are two general categories of methods for assessing the usability of a system: usability inspection methods and user studies. With usability inspection methods (such as cognitive walkthroughs and heuristic evaluations), evaluators inspect and evaluate usability-related aspects of a system. They are conducted without end users and require a certain level of expertise in usability [19]. These are useful in finding obvious usability problems but are no substitute for user studies with real users. Typically, usability inspection methods are used early on to guide the design process, then user studies are conducted to confirm the design decisions and find any problems that have been overlooked. User studies can range from closely controlled experimental studies testing specific hypotheses to field studies where the system is deployed for real usage and system logs and interviews are used to assess its usability. Most user studies fall somewhere in between, conducted in a lab, with pre-determined tasks, but also leaving room to observe users in a more ad hoc manner and uncover unexpected problems as they arise [27].

Usability tests are used to determine whether a system is suitable for the intended audience and for its intended purpose. Typically, the tests aim to uncover any difficulties encountered by the users as they go through a set of predetermined tasks. These tasks should be carefully chosen to reflect realistic usage scenarios. To preserve ecological validity, the environment should be set up to mimic reality as closely as possible in terms of technical details, but also in terms of instructions given. If a typical end-user is expected to be able to use the system without in-person training, then training on system use should not be provided during the test.

It is important to closely observe users as they perform these tasks as this is how most usability problems are revealed. The observer's role is mainly to observe and record what is happening. They need to be careful not to provide extra instructions or cues that may influence the user's actions. In fact, a script should be used to ensure that all participants receive the same information. It is important to emphasize that it is the system that is being tested and not the user; the participants should feel that they are helping with the development of the system rather than feel like their performance is being evaluated. A method called "think-aloud" is typically used, where users are asked to keep a running commentary as they perform the tasks. Pre/post questionnaires or interviews are also useful in gathering users' opinions, attitudes, and feedback about the system. This should be a secondary source of information, used in conjunction with observations and potentially system logs, because users' reported views often do not reflect their performance and often fail to reveal crucial usability problems.

Selecting the right participants for a usability study is important. The participants should accurately represent the users who would use the actual system, and be similar in terms of experience and knowledge. Improperly choosing participants will negatively affect the results of the tests, typically by missing critical usability problems.

The guideline stating that five users are enough to discover most usability problems [18, 29] has long been used to justify small usability studies. Recent work questions this assumption and highlights the fact that in most cases five users are not enough [9, 22, 28]. They found that some severe usability problems were only discovered after running a larger group of participants. The likelihood of finding usability problems is not evenly distributed. Some problems only arise under specific circumstances so using a small sample of users may not be sufficient to uncover them. The variability in the number of problems found by any one user also makes it unlikely that a sample of five users would discover most usability problems. Faulkner justifies that twenty users "can allow the practitioner to approach increasing levels of certainty that high percentages of existing usability problems have been found in the testing" [9].

3.2 Overview of Study

Our tests were conducted in Carleton University's Human-Oriented Technology Lab and the methodology was reviewed and approved by the university's ethics committee. Our study explicitly looks at the password managers as implemented rather than at the proposed additional implementations suggested by the systems' authors.

The typical tasks that users would need to accomplish with password managers fall into four categories:

1. Migrate user accounts (passwords) to use the password manager

2. Log in to protected user accounts from a primary computer
3. Change passwords for user accounts
4. Access user accounts remotely, i.e., from a computer other than the primary computer, such as on a public or friend's machine.

Each participant completed a one-hour session, where they completed a set of five tasks designed to simulate the real tasks that users would accomplish with the password managers. The set of tasks was repeated so that each participant completed them with both PwdHash and P-Multiplier. The order in which the tasks and the programs were presented was balanced to avoid bias. Throughout the session, the experimenter observed the participant and recorded their actions. Additional user feedback was gathered through questionnaires.

3.3 Participants

Twenty-seven adults participated in the study. Most were students at our university, from various faculties and degree programs; none were students specializing in computer security. A few had technical backgrounds: four were from Computer Science, one studied Information Systems, and none were from Engineering. Data from one participant was eliminated as a language barrier coupled with very little computer experience hindered their ability to understand the tasks. Of the remaining 26 participants, 21 were between the ages of 18 and 30 and five were over 30 years old. Data from these 26 participants was used for all further analysis in this paper.[2]

The participants were familiar with using the web and logging on to web sites requiring a username and password. All but two reported visiting the web daily, and these two said they were online several times a week. The participants were fairly comfortable with using computers; 24 of the participants self-rated their general computer skill level at 6 or higher on a scale of 1 to 10.

We chose not to screen participants based on experience using Firefox. Typical Firefox users are more technically sophisticated than average users so pre-selecting on this criteria would have biased our pool of participants. Additionally, interaction with the browser's interface was minimal; participants simply had to enter URLs and navigate within web pages. These tasks are accomplished in the same manner in Firefox and Internet Explorer.

A pre-task questionnaire was used to gain insight into the participants' initial attitude towards web security and passwords. They reported using an average of six web sites requiring a password to log in. A summary of their responses is presented in Table 1.

Table 1: Participants' initial attitude towards web security and passwords. Results represent the number of participants (out of 26) responding yes to each question. *An additional 27% (7) responded "somewhat".

Question	**Number of Users**	
Do you sometimes reuse passwords on different sites?	96%	(25)
Are you concerned about the security of passwords?	*58%	(15)
Criteria for choosing passwords:		
Easy to remember	69%	(18)
Difficult for others to guess	54%	(14)
Suggested by the system	0%	(0)
Same as another password	62%	(16)
Other	12%	(3)
Participation in online activities requiring personal or financial details:		
Online purchases	62%	(16)
Online banking	73%	(19)
Online bill payments	73%	(19)
Other activities	27%	(7)

3.4 Tasks

Participants completed a set of tasks using two different computers during the session. Both computers were running Windows XP and Mozilla Firefox. One system had the PwdHash plug-in (version 1.0 for Mozilla Firefox) installed while the second computer included the P-Multiplier plug-in (version 0.3 for Windows, Linux, and Mac OS).

The tasks are described in the following list. The *Second Login* task is dependent on the *Update Pwd* task, i.e., users must have successfully changed their password before they are able to log on to the site a second time with their new protected password. All other tasks are independent of each other. We did not include a "delete password" task because neither system supports this functionality. The tasks are:

Log In: Logging on to a web site that already has its password protected by the plug-in. This simulates how users log on once their passwords have been converted to protected passwords.

Migrate Pwd: Logging on to a web site with an unprotected password then changing the password so that it becomes protected. This is required by users to initially migrate each of their passwords.

Remote Login: Logging on to a web site with a protected password from a remote computer that does not have the plug-in installed. This models how users would log on to their accounts from a computer other than their primary machine.

Update Pwd: Logging on to a web site with a protected password then changing it to a new protected password. This situation would arise if users had to change their password once it is already protected.

Second Login: Logging on to a web site a second time, once the user has changed the password to a protected password. This task tests whether users understand how to log on to their account once they have changed to a protected password.

The tasks were set up using popular web sites (Hotmail, Google, Amazon, and Blogger) that users may encounter in real life. Test accounts were created so that participants did not use their personal accounts or passwords at any point during the experiment.

Participants completed the set of tasks with both plug-ins; the order was balanced so that each plug-in was seen first the same number of times. The order of the tasks within a set was also shuffled but an individual participant saw the tasks in the same order for both plug-ins. The *Update Pwd* and *Second Login* tasks were ordered so that they were always separated by exactly one task (for example, a participant completed the tasks in the order of *Log In*, *Remote Login*, *Update Pwd*, *Migrate Pwd*, *Second Login*). This ensured that participants changed their focus for a time before logging on to the web site a second time with their new protected password. One participant quit after completing the tasks only with PwdHash, but the remaining participants completed all tasks.

One of the difficulties with testing the usability of these plug-ins is that they initially have no visible interface. Even during the interaction, only P-Multiplier has a visible pop-up window. So simply giving the tasks to participants without instructions on how to use the plug-ins would have been futile. To preserve ecological validity, we tried to keep the instructions to a minimum; giving them written details of how to activate the plug-in, a brief explanation of how to change a password, and a short description of how to log on to a web site using a remote computer. The entire set of instructions was approximately half a page long for each plug-in (see Table 2). Users typically do not read manuals when they use software [3, 17, 31] so having participants follow detailed instructions would not have reflected a realistic scenario. Participants were also given a list of the usernames and passwords that they would require to complete the tasks. To minimize the effect of learning new passwords, a simple, one-word password was given for all tasks within a system ("alphabet" and "carleton"). These passwords were also written on a sheet in front of participants throughout the session.

Participants were given the instruction sheet for the particular plug-in and told that they could refer to it whenever necessary. They were directed to a computer with a Firefox browser window open and the appropriate plug-in pre-installed. They were instructed to pretend that this was their home computer and they should use Firefox as the browser for these tasks. Participants completed all tasks with a plug-in before switching computers to repeat the tasks with the second plug-in. No participant expressed any concern over using Firefox instead of the more popular Internet Explorer and no difficulties were observed due to using this alternative browser. Firefox was selected as the browser because the stable versions of the plug-ins were not available for Internet Explorer. Firefox was used in the original PwdHash usability study as well.

Table 2: Example instructions given to participants on how to use PwdHash

PwdHash Instructions: Add @@ in front of passwords you want to be made secure, this will activate PwdHash. PwdHash will transform the password before sending it to the web site. For example, if your password is "bob", enter "@@bob". You can also activate PwdHash by clicking on the password field and pressing the F2 key before entering your password. **To reset a password:** If your old password was not protected, enter the old password without activating PwdHash. When entering the new password, include the @@ at the front of the new password. This will activate PwdHash and transform this particular password. If your old password was already protected by PwdHash, activate PwdHash for your old password. When entering your new password, activate PwdHash and enter a new password for the site. **To use remotely:** To log in to a web account from a computer that does not have PwdHash installed, visit: http://crypto.stanford.edu/PwdHash/RemotePwdHash to generate your protected password. Enter the address of the target site and your password. The protected password will be generated. It can be copied/pasted into the password field of the target site.

Each task was described on an index card (see Table 3 for an example). The card also included two questions asking participants to rate the difficulty of the task and their satisfaction with the software for this particular task. Participants could take as long as they needed to complete the task and were told that if they felt they had spent enough time on a task and could not complete it, they could quit. At the end of each task, they circled their responses to the two questions and were provided with the next index card.

When participants reached the task where they had to log on to a web account from a remote computer (*Remote Login* task), they were instructed to change computers and pretend that they were now at their friend's house where the software was not installed. This proved problematic for P-Multiplier since the authors' solution to remote access is to install the plug-in. Participants could not install the plug-in on the second computer because it had PwdHash installed and the combination of the two crashed the computer. The *Remote Login* task was therefore eliminated for P-Multiplier. Judging from participants' reactions as they read from the instruction sheet that they had to install software for remote access,

Table 3: Example index card given to participants for the *Log In* task

Log on to www.google.com. Your password is protected by Password Multiplier.
This task was:

very easy	easy	neutral	difficult	very difficult

For this task, how satisfied are you with the software used to manage the password?

very dissatisfied	dissatisfied	neutral	satisfied	very satisfied

they would not have been pleased with this solution even if they had been able to complete the task.[3]

After completing a set of tasks, participants answered a paper questionnaire about their experience with the particular plug-in. The entire process was repeated for the second plug-in. A final post-task questionnaire asked participants to compare the two plug-ins.

3.5 Data Collection

Data was collected in two ways: through observation and through questionnaires. An experimenter sat with each participant throughout the session, recording observations, noting any difficulties, any obvious misconceptions in the participant's mental model of the software, any comments made by the participant, and whether they successfully completed the task. Participants were asked at the beginning of the session to "think-aloud". Besides the standard instructions given to all participants, no further explanations were given even if a participant asked for more instructions. In these cases, the experimenter remained cordial, clarifying that we were testing the usability of the systems and needed to see if people could use them without explanations. Occasional prompts such as "what did you expect to happen there?" were used if participants forgot to think-aloud.

The users' goal was to successfully complete the tasks using the given password manager. They were given as much time as they wanted and the observer waited for the participants to signal that they had completed the task or that they had run out of ideas and could not complete it. The outcome of each task was recorded by the observer according to the following possibilities:

Successful: The participant completed the task without difficulty.

Dangerous success: The participant eventually completed the task after several attempts (i.e., had difficulty). The negative impact is that in some cases, the unsuccessful attempts prior to the eventual success expose the password to attack (see Section 5.4).

Failed: The participant gave up on the task without completing it.

False completion: The participant failed to complete the task but erroneously believed that they had in fact been successful.

Failed due to previous: The participant could not complete the task because they had incorrectly completed the preceding task. This only applies to the *Second Login* task, where the *Update Pwd* had to be successful in order to proceed.

The first outcome is considered most positive. The second is somewhat positive but users may have exposed their passwords to danger (e.g., to JavaScript attacks and phishing) as they floundered with the task. They may even have inadvertently exposed multiple passwords, since a typical reaction to being unable to log in is to try all of one's passwords to see if something will work. The fourth outcome is especially dangerous because it leads to a false sense of security on the part of users.

Secondary measures taken in the study consisted of several Likert-scale questions [16]. These ask respondents to choose their level of agreement with the given statement from a set of possible answers, usually ranging from strong agreement to strong disagreement. We used a 5-point scale (strongly disagree, disagree, neutral, agree, strongly agree). Participants answered two of these questions on the index cards after each individual task, then completed a 16-question questionnaire for each plug-in.

The questions from the questionnaire were a priori grouped into four sets that considered different aspects of the interaction: perceived security, comfort level with giving control of passwords to a program, perceived ease of use, and perceived necessity and acceptance. Each set contained four similar questions (see Table 4); the questions were randomly organized on the questionnaire so participants were not aware of the groupings. Participants circled their answer for each question among the five choices. Half of the questions were inverted to avoid bias.

4 Collected Results

Neither PwdHash nor P-Multiplier fared well in terms of usability. Both the quantitative and observational data point to major problems, as explained below.

Our first measure of usability was whether participants were able to successfully complete the given tasks with each password manager. Our goal was not to provide a measure of how much better one password manager is compared to the other but to investigate the usability of

Table 4: Sample questions for each question set (for PwdHash, the questionnaire for P-Multiplier was identical other than the name of the software).

Perceived Security
My passwords are secure when using PwdHash.
I do not trust PwdHash to protect my passwords from cyber criminals.
Comfort Level with Giving Control of Passwords to a Program
I am uncomfortable with not knowing my actual passwords for a web site.
Passwords are safer when users do not know their actual passwords.
Perceived Ease of Use
PwdHash is difficult to use.
I could easily log on to web sites and manage my passwords with PwdHash.
Perceived Necessity and Acceptance
I need to use PwdHash on my computer to protect my passwords.
My passwords are safe even without PwdHash.

each. Looking at Tables 5[4] and 6, it appears that PwdHash outperformed P-Multiplier but still had a relatively high chance of potential security exposures, as many of PwdHash's successful outcomes were only realized after multiple attempts. These latter successful outcomes – labelled "dangerous successes" – can only be cautiously viewed as successes (see Section 5.4). The web sites used for the tasks were specifically chosen because they have a very high tolerance for incorrect login attempts. Participants frequently attempted to log in three to ten (or more) times before they were successful or gave up. With sites that limit the number of attempts, most users would have been locked out.

For the *Migrate Pwd*, *Update Pwd*, and *Second Login* tasks, a number of participants felt that they had successfully completed the task when in reality they had not. This was more common with P-Multiplier. This was mainly due to participants incorrectly believing they had successfully migrated their password from unprotected to protected and subsequently believing that they were logging on with a protected password when they were still using an unprotected password.

The Likert-scale responses from the questionnaires were converted to numeric values (1 = most negative, 3 = neutral, 5 = most positive). The responses were grouped according to their predefined sets to find the mean responses for each set. Means were calculated and differences between P-Multiplier and PwdHash were assessed by running t-tests. In a strict statistical sense, Likert-scale data should not be converted to numerical data. Since it is ordinal data, the differences between "strongly agree", "agree", and "neutral" are not necessarily the same. However, in practice this type of statistical analysis is the most common and accepted way of reporting Likert-scale data as the difference in results between parametric and non-parametric analysis are usually minimal.

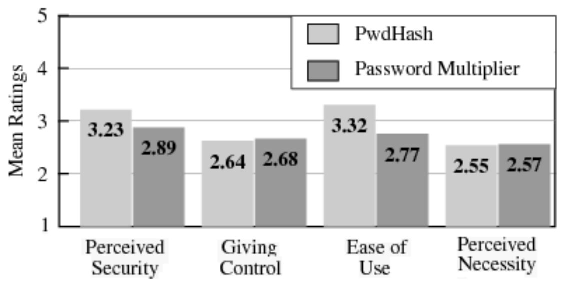


Figure 3: Mean questionnaire responses for each question group on scale of 1 to 5 (1 most -ve, 3 neutral, 5 most +ve)

We used t-tests to analyze the response distributions and determine the statistical significance of any differences. The t-tests can only be used to compare PwdHash and P-Multiplier against each other since we do not have an optimal system against which to compare the two. Examining the questionnaire data, the means for each group of questions reveal that neither systems fared very well; most values remained below neutral on the scale (see Figure 3). However, the t-test[5] showed that PwdHash was reported to be easier to use ($t(24) = 2.24$, $p < .05$) and perceived as more secure ($t(24) = 2.70$, $p < .05$) than P-Multiplier. The t-tests further revealed that the systems were similarly bad at making users feel comfortable with giving control to a password manager ($t(24) = -0.362$, $p = .721$) and that there was no difference between the two programs in how users felt regarding the perceived necessity of such systems ($t(24) = -0.207$, $p = .838$).

Examining their responses to the two questions from the index cards, we find that tasks completed with PwdHash were perceived as easier than those completed with P-Multiplier. Although participants reported higher satisfaction with PwdHash than P-Multiplier, in most cases the mean perceived difficulty and perceived satisfaction was below 4 for each. This means that participants initially reported positive reactions to the plug-ins. However these reported opinions need to be taken in context with user performance. In some cases, participants reported that the task was easy and that they were satisfied with the software even when they were unsuccessful at completing the task. In some of these instances participants were unaware that they had failed to complete the task. For example, they believed that they had generated a new secure password for a site when they had not even activated the plug-in – a potentially dangerous situation (see Section 5.4). In other cases, they said "well, this *should* have been easy, so I gave it a high rating". Obviously, relying solely on reported satisfaction and diffi-

Table 5: Task Completion Results for PwdHash

	Success		Potentially causing security exposures							
			Dangerous Success		Failures					
					Failure		False Completion		Failed due to Previous	
Log In	48%	(12)	44%	(11)	8%	(2)	0%	(0)	N/A	N/A
Migrate Pwd	42%	(11)	35%	(9)	11%	(3)	11%	(3)	N/A	N/A
Remote Login	27%	(7)	42%	(11)	31%	(8)	0%	(0)	N/A	N/A
Update Pwd	19%	(5)	65%	(17)	8%	(2)	8%	(2)	N/A	N/A
Second Login	52%	(13)	28%	(7)	4%	(1)	0%	(0)	16%	(4)

Table 6: Task Completion Results for P-Multiplier

	Success		Potentially causing security exposures							
			Dangerous Success		Failures					
					Failure		False Completion		Failed due to Previous	
Log In	48%	(12)	44%	(11)	8%	(2)	0%	(0)	N/A	N/A
Migrate Pwd	16%	(4)	32%	(9)	28%	(7)	20%	(5)	N/A	N/A
Remote Login	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Update Pwd	16%	(4)	4%	(1)	44%	(11)	28%	(9)	N/A	N/A
Second Login	16%	(4)	4%	(1)	16%	(4)	0%	(0)	64%	(16)

culty is misleading.

At the end of the session, participants were asked which of the two systems they preferred. Participants were nearly evenly distributed in terms of preference: 14 participants selected PwdHash and 11 chose P-Multiplier. The total number of responses is 25 because one participant only completed tasks with one system and could not compare the two.

5 Analysis and Interpretation of Results

Only one task in our study had a success rate of over 50%. This should concern the security community because when users cannot use a system correctly, they become vulnerable to attacks (see Section 5.4). It is important to examine the causes of failure in order to learn how to address these usability problems. The best source of information in this case is the observational data that recorded what happened as participants tried to use the systems.

Section 5.1 reports on usability problems common to both systems tested. Section 5.2 compares our findings with those of PwdHash's authors and section 5.3 reports on usability problems discovered specifically in P-Multiplier. Finally, section 5.4 discusses the particular security vulnerabilities exposed due to these usability problems.

5.1 Problems Common to Both Systems

Multiple issues arose because users' mental models did not match the reality of the system. They clearly were trying to make sense of what they saw and experienced during the interaction, but their understanding was incomplete or incorrect. Specifically, they had difficulty understanding when and how to activate each system, understanding how long it remains active once it is activated, determining to what fields the activation applied, and determining whether they had correctly accomplished a task.

Users were unsure about whether the systems were correctly activated. They often commented on "well, I *think* it did something" or "I guess that's what needed to be done". They perceived little feedback and were looking for some cue that they had been successful. One participant somehow decided that the "lock" icon on the browser that indicates whether the site is secure was the indicator of whether the password was protected. In each task, they looked at the icon to make sure it was closed then happily entered their password without activating the plug-in, fully believing that their password was protected. Another participant who could not figure out how to activate the plug-in reasoned "this password must be *really* secure – I can't even get in".

Another misconception was that they could activate the password manager once and it remained active throughout their computer session. They double clicked or pressed @@ with the very first password they entered and then assumed that all further passwords from this point onwards were protected without further action. This raises serious concerns because it gives users a false sense of security. They believed they were protected while in fact their weak passwords continued to be used for their accounts. They were able to log in to their accounts because they never actually converted their passwords to "protected" passwords even though they believed that they had. In some cases this might possibly lead to even weaker password choices than normal because users believe they are being protected.

A second activation problem arose with each password managers' "alternative" trigger mechanism: pressing the F2 key for PwdHash and Alt+P for P-Multiplier. Both of these required that the users' cursor was already in the password field before triggering the program. Users would forget to click on the password field, then incorrectly assume that the program had been activated when they pressed F2 or Alt+P.

Several users erroneously believed that unique, random passwords were generated for them each time they activated the password manager; even for the same web site. For example, they believed that each time they logged on to Hotmail and used the password manager, a new, unique password was being generated. Of course this would not work since a web site expects the same password each time in order to authenticate the user. But this view was even held by participants who would be considered advanced or expert computer users such as Computer Science graduate students.

Not all usability problems encountered were a direct result of the password manager interfaces. Some problems were due to bad web site design. The sites used in the study were popular sites frequented by expert and novice users alike so the observed problems are likely to occur in real life as well. Participants had difficulty finding the login areas, had difficulty finding where to change their passwords, had difficulty changing their passwords, and had difficulty determining if they were correctly logged in to a site. These are valid usability issues that provide context and insight into the circumstances and environments where people will be using password managers. They must be taken into account even though they are not a direct result of the password manager interface. Another problem noticed by several participants was the inconsistency in designating the username field: it was called "username", "account name", or "email" on different web sites. The instruction sheet referred to it as "username"; this was sufficient to raise several questions.

Several participants gave up on tasks out of frustration, especially with P-Multiplier. Most said that in real life, they would have requested that their password be reset or would have created a new web account by this point. None mentioned that they would have looked for further documentation. Some assumed that something was wrong with the account and asked the observer to correct it, while others apologized for their "stupidity" and blamed themselves for the problem.

Another frustration shared by many participants was that they did not know their "actual" passwords. A second subset never even realized that this was in fact what was happening, revealing further mental model issues. Of those who understood this concept, they felt that the program should "trust" them with their own passwords: "why won't it tell me my password?" and "I wish it would show me my password when it first generates it, I won't lose it or share it!". Presumably, the tendency would be for users to write down their passwords for safekeeping; this has advantages and disadvantages, depending on the threat model.

Users also did not like relinquishing control to a computer program: "How do I know I can trust it?", "Now I have to trust two companies with my passwords, who guards the guards?" and "I felt like I had no control. Passwords are important but I have no idea what happened." Even though they know that password security is a problem, they feel that they are best equipped to care for their own passwords. This view is clearly contrary to that of security experts. The mismatch needs to be addressed if widespread adoption of security measures is to take place.

5.2 Comparison of findings with previous PwdHash findings

The authors of PwdHash briefly report the findings of their user study as follows [24]:

- *"The participants did not experience any major difficulties signing up for new accounts and logging in to them using the password prefix."*
- *"The user interface was so invisible that many participants did not observe the extension doing anything at all. They did not notice the lock icon, and their only clue that the extension was working was the fact that their password changed length when focus left the field, which they found confusing"*
- *"It was only once the users had to log in using a different browser that didn't have PwdHash installed that they encountered difficulties. They found the process of copying over the site address into the remote hashing page to be annoying, and if they did so incorrectly (for example, using gmail.com instead of google.com) the site that they were logging into did not provide useful feedback as to what went wrong."*
- *"When presented with a fake eBay site at a numeric IP address, most of the participants were willing to try logging in anyway, but their use of the password-prefix prevented the phishing page from reading their eBay passwords."*

Very little detail is given about the user study, its methodology, its participants, or the actual results. This makes it difficult to assess the validity of the study and its thoroughness, or to attempt a replication study. We address each of their reported results by comparing them to our findings and report additional usability problems that we discovered. Our results contradict those found by the authors [24] with respect to the usability of PwdHash. We found that even though performing better than P-Multiplier, PwdHash still had major usability problems.

While the success rates for our tasks with PwdHash were fairly high, more than half of these successful outcomes were only achieved after repeated attempts and errors by the users. Many successes were due to participants trying random actions until eventually something worked, not because participants had a clear understanding of how to use the software.

The majority of our participants did not comment on the changing length of their password. Many did, however, fail to realize that they needed to enter @@ in front their new password when re-typing it in for confirmation on a Change Password page. They assumed that entering the @@ once was enough. Visually, it was apparent that the two new passwords did not match due to the length difference, but users did not notice this cue. Many were confused by the lack of feedback and did not know if they had been successful in activating PwdHash. A similar problem occurred with P-Multiplier where users would activate the plug-in the first time they entered their password then assumed that the program remained active and proceeded to re-type their password in the password confirmation field without re-activating P-Multiplier.

Most of our users also had difficulty with the remote interface, but the problems encountered were different from those reported in the original study. First, users had difficulty reaching the remote web site due to its complicated URL. Several commented that they "would never find this site when they needed it". The second concern cited by several users was that the remote web site did not ask them for a username. They found this disturbing and could not understand how it was going to generate the correct password when it did not know who they were. "How will it know to generate *my* password?" and "How does it know who I am?" were common questions. One shrewd participant eventually concluded that "wait, it's going to give anyone who enters my regular password the same complicated password? That's not good!". Users obviously believe that the generated passwords are (and should be) somehow unique to them. This points to problems with users' mental model of the systems.

The users who successfully completed the remote login task did not have any trouble identifying the correct site URL to enter (granted, we were using www.amazon.com as the site, alternatives may not have been as obvious to users as with the Google and Gmail problem reported in the first study). Of those who failed to complete the task, most never even reached the remote web site. Although they were explicitly told that "you are now at your friend's house, they don't have the software installed, so you will need to log in remotely", they still attempted to log in using the @@. When this failed, they attempted to use their plain password and made random attempts at guessing the correct password. They did not refer to their instruction sheet that clearly had a section entitled "To log in remotely:". Even when users had only half a page of instructions, directly in front of them, they tended not to refer to it. On a similar note, only one user actually read and commented on the instructions posted on the remote web site about whether to enter @@ in front of their password on the remote site. Without reading the instructions, about half of the users entered their password with the @@ and half omitted it. Apparently, the software is capable of dealing with both situations because they were successful in generating their password in both cases. This may be good in terms of anticipating users' actions but breaks the mental model of what the @@ symbols represent.

We did not test if users would try to log in to a fraudulent site. Based on our observations, we predict that users would attempt to log on without realizing the danger. And based on users' behaviour at other sites when they had an incorrect password, it is our guess that users would divulge their plain text password when the first log in attempt failed and they tried alternatives. Thus even though PwdHash would protect their initial attempt, users would provide their clear text password on subsequent attempts thus creating a security vulnerability.

PwdHash alerts users when it thinks they are trying to enter a password into a non-password field. The alert message is long; most users simply dismissed it without reading it. Compounding the problem is that this alert sometimes appeared with legitimate password fields, usually when a user would start typing a password and then try to include the @@ as an afterthought.

PwdHash more closely mimics the interaction with which users are familiar. From a conceptual point of view, users simply have to remember that passwords starting with @@ are protected. They do not need to really understand that the @@ is invoking a plug-in that performs a cryptographic hash on their password. On the other hand, this familiarity also caused confusion. One participant voiced this concern: "Really, I don't see how my password is safer because of two @'s in front". This user obviously understood that the @@ were important but could not make the connection with typing in these symbols and triggering a program that will generate a stronger password. Some people were uncomfortable with the transparency, commenting that "at least with P-Multiplier I get a window, the password is going into the program".

5.3 Comments on usability of Password Multiplier

Although no user study was documented, we now comment on statements made in Halderman et al. [11] about the usability of P-Multiplier. They argue that their approach is *"both convenient for users and highly secure, a combination not offered by previous designs"*. In terms of convenience, they claim that their software can be run on different machines to access their password from anywhere and that the memory load is light because users only need to remember a username, the domain name for the site being accessed, and a single master password. They also highlight their browser integration, stating that this integration *"allows it to work as conveniently and transparently as possible, minimizing the burden imposed on the user and increasing the chances that the system will be used in practice. Second, browser integration allows our system to offer some protection against spoofing and phishing attacks, since by default the password program will use the name of the server that will actually receive the submitted password, even if an attacker has tricked the user into believing she is connecting to a different site"* [11].

We argue that P-Multiplier fails to meet the usability goals its authors set for themselves. P-Multiplier is not easily portable since it requires that the program be installed on a remote computer. Many users do not have privileges allowing them to install software on such machines and would likely find it inconvenient to install software in order to log in to a web site.

The memory load is not trivial. Users must remember the master username and master password for P-Multiplier. In addition, they must still remember their username for each web site. And finally, they must remember the modified domain names for all sites where they have changed their password.

Transparency is often a cited goal for security systems. However transparency is often translated to a lack of feedback and this leads to many usability problems as users are unable to form an accurate mental model of the system and its operation. Two participants never realized that they only needed one password with P-Multiplier. Each time, they entered a clear text password in the web site's password field, double clicked to activate the dialog box and then entered the master password. They never realized the generated password was overwriting their clear text password and that they did not have to manually enter anything into the web site's password field.

P-Multiplier's authors state that their plug-in helps to protect against phishing attacks. While this is true, it relies on the assumption that users correctly use the system. This is a problematic assumption, as from our observations, users frequently entered their plain text passwords and resorted to random guessing because they could not correctly use P-Multiplier. In these cases, users are no more protected against phishing than they would be without a password manager. (Some users might possibly be less diligent in scrutinizing sites because they assume that the software is protecting them.)

Users had high expectations of the password managers. They expected that any password generated by the software should be extremely strong, regardless of their own input. One web site offered a "strength-meter" when changing passwords, rating the strength of new passwords. When the password manager failed to generate a password considered "strong" by the web site, users expressed their concern and disapproval. Specifically, passwords generated by P-Multiplier were often rated as "medium" by Hotmail rather than "strong". Users felt that if they chose their own passwords, they would be able to produce a strong password.

P-Multiplier's solution for changing a password once it is protected is nearly impossible for users to understand. Of those who did understand it, they expressed frustration at now having to remember the modified domain and having to enter it each time they logged in to this web site. There also seems to be a bug in the program where users were able to change their password by entering a different master password instead of changing the domain. The plug-in accepted this incorrect master password and somehow generated a new password for the site. It was unclear whether the passwords so generated were secure, and it violated the user's mental model that all passwords in P-Multiplier are protected by a single master password.

5.4 Usability problems causing security exposures

Usability problems are of general concern in HCI and should also concern the security community because they may also lead users to bypass security mechanisms. Here, we comment on a separate matter: usability problems which may directly cause security exposures, even when the user has the intention of complying with the security mechanism.

The first concern is users failing to properly activate the respective mechanisms (e.g., failing to enter @@ or double-click the web site password field). For both password managers, when users enter their password but forget to invoke the mechanism, the (raw) password is sent on as if the manager software was not installed, naturally exposing it to JavaScript and phishing attacks – both of which PwdHash was designed to counter. In the case of P-Multiplier, this exposes what is the master password, sufficient to generate site passwords for many sites. This user error is a factor in three possible failure outcomes (Dangerous Success, Failure, False Completion).

The same problem exists when users change passwords. The interface requires entering the old password, the new password, and re-entering the latter. In some cases, users activate the respective mechanism for the first new password field, but then re-enter their password the second time *without* activating it.

A different problem resulted from user confusion. When users failed to correctly activate the mechanism, some started guessing, entering all passwords they could think of, with and without the activation sequence. In this case, a phishing site (or JavaScript attack) could harvest a number of passwords of interest.

Another generic danger, not specifically related to the password managers in question, is that users who believe a password manager mechanism will be used may be more inclined to choose a simple (low-entropy) password, believing it will be strengthened; when it is not, it may be even more vulnerable to dictionary attacks.

6 Further Discussion and Recommendations

As discussed in Section 1, the problem of establishing and managing strong passwords for large numbers of accounts is unlikely to disappear soon. While password managers offer a solution, current implementations suffer from significant usability problems, which in our observation fall into two main categories: (1) users' mental models; and (2) users' views on the necessity of tools such as password managers, and acceptance (their willingness to hand over control of passwords to a computer program). We discuss these, in turn, in Sections 6.1 and 6.2. In Section 6.3, we consider additional criteria for security software to be usable.

6.1 Mental Model

Incorrect user mental models appears to be the biggest problem with the two password managers tested. While manifested in different ways, the common cause is that users do not even have a high-level understanding of what the manager programs are doing. The importance of a mental model for usability in security is a key issue identified by Whitten and Tygar [31], and as early as 1975 by Saltzer and Schroeder [25].[6] To be usable, a system must support a user's mental model, and must fit into their regular work patterns. This is not simply a matter of user satisfaction; failure to do so can result in security exposures. Through interactions with a system, users form a mental model as a mechanism to help in understanding, learning, and remembering. The primary means of doing so is by interpreting the actions and interface of the system as they are presented; an incoherent or inappropriate presentation leads to an erroneous, or incomplete model. A user's model serves as the basis for their interactions; thus improper models often lead to major usability problems. Norman's [21] Gulf of Evaluation is a measure of how much effort must be exerted to determine whether the system actions correspond to the user's intentions and expectations. Ideally this gulf should be small, indicating that the system provides accurate, understandable, and visible feedback to the user.

It is the responsibility of a good designer to ensure that users have the cues needed to form an accurate and complete mental model of a system [20]. Although the usability literature has long supported the need for accurate mental models, this seems not yet to be common knowledge – or perhaps more correctly, not common practice – for the security community. Users need not fully understand the details of complex security programs, but rather need a mental model that is consistent, and that will allow them to predict program behaviour and the results of their own actions.

In addition, the interfaces should provide better feedback. It is a well accepted usability principle that system feedback is necessary to narrow the Gulf of Evaluation and for users to develop accurate mental models. In our study, users were often left in states where they could not tell if their actions had been successful. In some cases, the systems provided feedback but it was not noticed by the users. P-Multiplier changed the background colour of the password fields containing generated passwords, but in our study not one user remarked on this subtle change or appeared to have noticed it. PwdHash alerted users when they tried to enter a password in a non-password field. However this alert sometimes appeared when users were in a valid field and its wording was confusing for users. Most users simply dismissed the warning. Participants were also confused when something had obviously gone wrong, but they received no feedback as to what may have caused the problem or how to recover from it. In their effort to be unobtrusive and subtle, these interfaces have reached a point where they are impossible to comprehend due to a lack of cues.

We make the following suggestions regarding feedback provided to users by the password managers tested:

1. It should be obvious when a password has been protected.
2. It should be obvious when the plug-in has been activated and is awaiting input. This directly contradicts the assumption that transparency is good for security interfaces. The lack of visual cues was problematic for both programs because it left users confused and unsure about how to proceed.
3. It should be clear how existing passwords are migrated (from pre-manager unprotected, to with-manager protection). Even with instructions on how to activate the program and an explanation that users must change their password, confusion arose. Several users attempted to "change" their password at the initial login prompt for a webpage rather than logging on then using the site's Change Password interface. They felt that since the program was installed on the computer, "changing their password" meant that they could simply start using the plug-in to enter their password on a web site.
4. If something goes wrong, feedback should be short, understandable, and reveal how to address the problem. This is a standard usability principle. Unfortu-

nately, it is unclear if this would be easy to implement since the problem may stem from the target web site rather than from the plug-in.

5. There should be a way for users to check which of their accounts are currently protected. Migrating to one of these systems is non-trivial since users will need to track which accounts have been migrated and which remain to be done. We suggest that the plug-in keeps a list of currently protected web accounts on the user's primary computer.

Better integration with the actual web pages may be technically difficult, but it would certainly help users. For example, when a password is incorrectly entered, users currently do not know if the problem is a typing error or an error in activating the manager. We observed users resorting to random guessing in hopes that something would work – with one security risk being that such passwords could all be exposed (see Section 5.4) and might include sensitive passwords (which the user resorts to trying) for unrelated accounts. Providing accurate error messages would reduce user frustration.

6.2 User Acceptance and View of Necessity

From a usability standpoint, user satisfaction and acceptance is always important (although sometimes users may have no choice, e.g., when it is required for a critical part of their job). When users must make an "opt-in" choice, it is particularly important that users accept the system – otherwise they may turn to an alternative (possibly insecure) service or find ways to bypass the security mechanisms [2, 5, 26]. Lack of user satisfaction and acceptance can lead to a lack of security.

We believe that helping users form a clearer mental model would significantly help with user acceptance of the studied password managers. Currently, users are uncomfortable with using the software (see Figure 3) and do not trust it because they do not understand it. They are worried about the safety of their accounts. They are worried that they will be unable to reach their accounts because the password manager will stand in their way. As an intermediary, the password manager needs to appear reliable, consistent, and predictable.

To increase user acceptance, we also recommend that along with the installation of password managers, users be educated about the importance of protecting passwords, and how password managers can achieve this goal. As with other security measures this is not a simple task since security is not the primary goal of most end-users; however, password managers have the advantage of potentially simplifying users' tasks (e.g., by requiring them to remember fewer or less-complicated passwords). Once users understand this, we would expect higher user acceptance.[7]

6.3 Criteria for Security Software to be Usable

Whitten and Tygar highlighted issues that arise when users have inaccurate or incomplete mental models, suggesting that for *security* software to be usable, users must [31]:

1. be reliably made aware of the security tasks they must perform;
2. be able to figure out how to successfully perform those tasks;
3. not make dangerous errors; and
4. be sufficiently comfortable with the interface to continue using it.

We suggest the following two additional criteria (closely related or supporting 2 and 3):

5. **be able to tell when their task has been completed**; and
6. **have sufficient feedback to accurately determine the current state of the system.**

The fifth concerns a usability problem seen in both the Whitten and Tygar study and our current study: users were unable to tell whether their task had been successfully completed and sometimes incorrectly assumed success. This can cause security vulnerabilities (e.g., as information believed to be secure can be left unprotected). The sixth draws on the well-known usability guideline of feedback, which is especially important for supporting accurate mental models in security interfaces. Transparency in this case can be dangerous because it leaves users free to make assumptions about the system that could lead to security exposures.

7 Related Work

Background on usability testing is given in Section 3.1. Section 2 mentions a few alternate password managers (see [11] and [24] for a good summary of other password managers). Here we focus on related work including usability tests for authentication mechanisms. Although the situation is now changing significantly, there have been surprisingly few such academic papers.

Growing interest is reflected by Cranor and Garfinkel [4], and the Symposium on Usable Privacy and Security (SOUPS). Zurko and Simon [34] introduced "user-centered security" in 1996. Prominent among past work is the case study of PGP 5.0 [31], which included a cognitive walkthrough inspection analysis and a lab user test involving 12 participants (see Section 6). Another early authentication usability study is the Déjà Vu work [8], which included interviews with 30 people on password behaviour, and user testing with 20 participants; the focus of the user testing was on creation of password (image) portfolios, and memorability results. Prior to this, Adams and Sasse [1] explored password-related user behaviours and memorability issues through questionnaires and interviews, leading to a number of recommendations.

Recent papers involving user studies on graphical passwords include Davis et al. [6] with focus on security and poor user choices made by real users; and Wiedenbeck et al. [32] with focus on memorability in the PassPoints system, presenting results of a user study involving 32 undergraduates. Weinshall [30] introduces a challenge-response authentication protocol relying on recognition of images and presents results of a small user study.

Although not specifically on passwords, Garfinkel and Miller [10] carried out a 43-subject user test of a secure email prototype with focus on key continuity management features (automating certain key and certificate management activities related to signing email); they report increased protection against certain forms of social engineering, but not from attacks from new (unfamiliar) email addresses or from phishing.

Related to our observations that more visibility (vs. more transparency) would enhance usability of some security features, Depaula et al. [7] explore making relevant features of security mechanisms – including configurations, activities and implications of available security mechanisms – visible, to allow more informed user decisions.

8 Concluding Remarks

While the security community seeks to develop systems with stronger security, it is now commonly recognized that even the most technically secure system, if unusable, will fail in practice. However, without measurements on real users, we cannot evaluate usability. Usability tests with real users should be included in not only the development of security systems, but also in research which proposes new security tools and plug-ins. Both formative and summative usability tests are desirable. Formative tests are conducted throughout the development of the system to guide the development and find potential usability problems as they arise; these tests are typically less formal and their goal is to highlight any problems. Summative tests are used to gather performance data and provide measures of usability; they are more controlled and their goal is to validate the usability of a system or compare its performance for different groups of users.

We have refrained from making specific suggestions for changing the interfaces of the studied password managers, as any suggestions should themselves be tested for usability – and we have not done so. Thus until such time, we cannot authoritatively conclude that they would work. Instead, we have suggested further guidelines and requirements for the interfaces. Further work is needed to identify specific mechanisms to use in order to comply with these guidelines and address the requirements.

The goal of usability studies is to uncover problems so that they can be corrected. We have identified several usability problems with Password Multiplier and PwdHash which we believe are likely to exist in other similar password manager proposals. The next step is to identify mechanisms, if possible, by which these interface problems can be addressed. Ideally, we would then build a new interface that implements these mechanisms, and conduct further usability evaluation to test if this actually improves usability.

9 Acknowledgements

We thank the anonymous reviewers for their comments which helped improve this paper to its present form. We also thank the members of the Carleton's Digital Security Group and Mary Ellen Zurko for their feedback on earlier versions of this work. The first and third authors are supported in part by the "Legal and Policy Approaches to Identity Theft" project funded by the Ontario Research Network for E-Commerce. The second author is Canada Research Chair in Network and Software Security, and is supported in part by the Canada Research Chairs Program, and an NSERC Discovery Grant.

Notes

[1]Lack of feedback hindered users' ability to form accurate mental models, and to determine if passwords were being protected.

[2]One person did not try P-Multiplier; they quit after completing the tasks with PwdHash. Therefore only partial data is available for this participant. This left 25 participants for P-Multiplier.

[3]Using a third computer would have been a better experimental design, allowing participants to complete the task, but we do not expect that this would have led to different results.

[4]Technical problems caused one participant to miss the *Log In* and *Second Login* tasks with PwdHash

[5]A t-test is a ratio giving a measure of the difference between two means relative to the variability of each set. Larger ratios mean that the two groups are more distinct from each other. Significance *p* shows the likelihood that the results are due to chance.

[6]They note [25]: "Psychological acceptability: It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly. Also, to the extent that the user's mental image of his protection goals matches the mechanisms he must use, mistakes will be minimized."

[7]Ideally, this hypothesis would be verified by a separate study.

References

[1] A. Adams and M.A. Sasse. Users are not the enemy. *Comm. of the ACM*, 42(12):41–46, 1999.

[2] R. Anderson. Why cryptosystems fail. In *Proceedings of the 1st ACM Conference on Computer and Communications Security.*, December 1993.

[3] J.M. Carroll, P.L. Smith-Kerker, J.R. Ford, and S.A. Mazur-Rimetz. The minimal manual. *Human-Computer Interaction*, 3:123–153, 1987-1988.

[4] L.F. Cranor and S. Garfinkel. *Security and Usability: Designing Systems that People Can Use*. O'Reilly Media, edited collection edition, 2005.

[5] D. Davis. Compliance defects in public key cryptography. In *Proceedings of the 6th USENIX Security Symposium*, July 1996.

[6] D. Davis, F. Monrose, and M. Reiter. On user choice in graphical password schemes. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[7] R. DePaula, X. Ding, P. Dourish, K. Nies, B. Pillet, D. Redmiles, J. Ren, J. Rode, and R. Silva Filho. Two experiences designing for effective security. In *First Symposium on Usable Privacy and Security (SOUPS 2005)*, Pittsburgh, July 2005.

[8] R. Dhamija and A. Perrig. Déjà Vu: A User Study Using Images for Authentication. In *Proceedings of the 9th USENIX Security Symposium*, 2000.

[9] L. Faulkner. Beyond the five-user assumption: Benefits of increased sample sizes in usability testing. *Behavior Research Methods, Instruments, & Computers*, 35(3):379–383, 2003.

[10] S.L. Garfinkel and R.C. Miller. Johnny 2: A User Test of Key Continuity Management with S/MIME and Outlook Express. In *First Symposium on Usable Privacy and Security (SOUPS 2005)*, Pittsburgh, July 2005.

[11] J. Halderman, B. Waters, and E. Felten. A convenient method for securely managing passwords. In *Proceedings of the 14th International World Wide Web Conference*, 2005.

[12] J. Alex Halderman. Password Multiplier web site, www.cs.princeton.edu/˜jhalderm/projects/password/, accessed January 2006.

[13] A. Karp. Site-specific passwords. Technical report, Hewlett-Packard Laboratories, January 2002.

[14] J. LaPoutre. Password Composer web site, http://www.xs4all.nl/˜jlpoutre/BoT/Javascript/, accessed January 2006.

[15] Password Maker web site, http://passwordmaker.org/, accessed January 2006.

[16] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 140, June 1932.

[17] B. Myers. Why are human-computer interfaces difficult to design and implement? Technical Report CMU-CS-93-183, Carnegie Mellon University, Department of Computer Science, 1993.

[18] J. Nielsen. *Usability Engineering*. Boston: AP Professional, 1993.

[19] J. Nielsen and R.L. Mack. *Usability Inspection Methods*. John Wiley & Sons, Inc, 1994.

[20] D.A. Norman. Cognitive engineering. In D.A. Norman and S.W. Draper, editors, *User Centered System Design: New Perspectives on Human-Computer Interaction*, chapter 3, pages 31–62. Lawrence Erlbaum Associates, Publishers: Hillsdale, NJ, 1986.

[21] D.A. Norman. *The Design of Everyday Things*. Basic Books, 1988.

[22] C. Perfetti and L. Landesman. Eight is not enough. *User Interface Engineering*, 2001.

[23] K. Renaud. Evaluating Authentication Mechanisms. In L.F Cranor and S. Garfinkel, editors, *Security and Usability*, chapter 6, pages 103–128. O'Reilly Media, 2005.

[24] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. Mitchell. Stronger password authentication using browser extensions. In *Proceedings of the 14th USENIX Security Symposium*, Baltimore, August 2005.

[25] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[26] M.A Sasse and I. Flechais. Usable Security: Why do we need it? How do we get it? In L.F. Cranor and S. Garfinkel, editors, *Security and Usability*, chapter 2, pages 13–30. O'Reilly Media, 2005.

[27] B. Shneiderman. *Designing the User Interface*. Addison Wesley, 3rd edition, 1998.

[28] J. Spool and W. Schroeder. Testing web sites: Five users is nowhere near enough. In *Proceedings of ACM Conference on Human Factors in Computing Systems (CHI 2001)*, 2001.

[29] R.A. Virzi. Refining the test phase of usability evaluation: How many subjects is enough? *Human Factors*, 34:457–468, 1992.

[30] D. Weinshall. Cognitive Authentication Schemes Safe Against Spyware (Short Paper). In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2006.

[31] A. Whitten and J.D. Tygar. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. In *Proceedings of the 8th USENIX Security Symposium*, Washington, D.C., August 1999.

[32] S. Wiedenbeck, J. Waters, J.-C. Birget, A. Broditskiy, and N. Memon. Authentication Using Graphical Passwords: Effects of Tolerance and Image Choice. In *First Symposium on Usable Privacy and Security (SOUPS 2005)*, Pittsburgh, July 2005.

[33] N. Wolff. Password Generator web site, http://angel.net/˜nic/, accessed January 2006.

[34] M.E. Zurko and Richard T. Simon. User-centered security. In *Proceedings of the 1996 New Security Paradigms Workshop*, pages 27–33, Lake Arrowhead, CA USA, 1996. ACM.

On the Release of CRLs in Public Key Infrastructure

Chengyu Ma[1]
Beijing University
machengyu@pku.edu.cn

Nan Hu, Yingjiu Li
Singapore Management University
hunan,yjli@smu.edu.sg

Abstract

Public key infrastructure provides a promising foundation for verifying the authenticity of communicating parties and transferring trust over the internet. The key issue in public key infrastructure is how to process certificate revocations. Previous research in this aspect has concentrated on the tradeoffs that can be made among different revocation options. No rigorous efforts have been made to understand the probability distribution of certificate revocation requests based on real empirical data.

In this study, we first collect real empirical data from VeriSign and derive the probability function for certificate revocation requests. We then prove that a revocation system will become stable after a period of time. Based on these, we show that different certificate authorities should take different strategies for releasing certificate revocation lists for different types of certificate services. We also provide the exact steps by which certificate authorities can derive optimal releasing strategies.

1 Introduction

The introduction of world wide web technology has resulted in a faster and easier exchange of information. It also exacerbate the problems of verifying the authenticity of communicating parties and transferring trust over the internet. The public key infrastructure (PKI) has been considered as a promising foundation for solving these problems, especially in the context of secure electronic commerce. Since the authenticity of PKI is achieved through the verification of digital certificates, it is crucial to understand the nature of digital certificates in practice.

Digital certificates have been supported by a wide range of entities. For example, Korean Government invested heavily on promoting digital certificates to the public. The digital certificates have been issued for various applications such as internet banking, government e-procurement and stock exchange.[2] In the year 2001, the Ministry of Information and Communication announced that the total number of users of PKI would reach to 10 million by the year 2002. Korea also made its own 128 bit encryption algorithm called SEED and encouraged all financial services to use it. It also developed a national certificate system based on public key infrastructure.[3]

Unfortunately the glory of the PKI can be so dimmed if there is no efficient way to verify the validity of digital certificates. Checking the authenticity and expiration date of a digital certificate is never sufficient enough as it is possible that a certificate has been revoked before its expiration for various reasons, such as 1) key compromise, 2)certificate authority (CA) compromise, 3) affiliation change, 4) superseded, or 5) cessation of operation [6]. To make PKI a useful platform, it is critical to manage the certificate revocations efficiently.

Previous research has concentrated on the trade-offs that can be made among different revocation options [6, 13, 15]. The purpose is to see which revocation mechanism is more efficient in which scenario. In order to compare the performance of different mechanisms, people ran simulations based on theoretical assumptions. For example, Naor and Nissim calculated the communication cost by assuming a fixed length of certificate revocation list (CRL). Cooper [2] and Arnes [1] modeled the distribution of revocation information by assuming an exponential inter-arrival probability for the requests for CRLs. To the best of our knowledge, no rigorous efforts have been made to understand the probability distribution of certificate revocation requests based on real empirical data.

Another key conclusion of previous research is that CA should release CRLs at a fixed time interval because consumers may not need the most current CRL. As long as a user has a CRL that is recent enough to meet its operational requirement, it is acceptable in practice [8]. Rivest [13] has proposed that the recency requirement should be set by customers, rather than CAs. Unfortunately, all the conclusions are based on some theoretical arguments, there is a lack of guidance for CA operations. Given a recency requirement to be set by the consumers, CA must understand the following aspects for setting an optimal CRL releasing policy: 1) Why CA needs to follow a given time interval? Will the interval be the same for a new type of certificates versus a type of certificates that has been provided by CA for a certain period of time? 2) Will the interval be the same for a mature CA versus a Start-up CA? 3) If CA does follow a given CRL releasing interval, how does it know whether that interval is optimal or not?

In this paper, we study how often should a CA release its CRLs. We concentrate our analysis on CRL because it is the most common and simplest method for certificate revocation [6]. We have several interesting findings: 1) Contrary to the common sense, the probability that a certificate being revoked is a decreasing function over the certificate's life cycle. People tend to think this probability either flat over time (as memory-less Poisson) or increasing over time. 2) CA should take different strategies for publishing certificate revocation lists when dealing with a new type of certificates versus a re-serving type of certificates. 3) A mature CA and a start-up CA should also take different strategies for releasing CRLs. 4) We give an optimal releasing interval prescription for CA to balance the trade-off between cost and risk. In a very general case, a mature CA who deals with just one type of certificates can save almost 40,000 dollars more over one-year operation if it follows our strategy by deceasing its CRL releasing interval from 34 days to 17 days.

The rest of the paper proceeds as follows: First, we briefly review the major concerns about the public key revocation. We then discuss empirically how to collect data and derive the revocation distribution. Next, based on the empirical distribution, we give CA an optimal CRL releasing strategy. Finally we conclude our paper with a discussion of contributions, limitations, and future research directions.

2 Literature Review

Since its introduction, the public key infrastructure [5] has provided a promising foundation for verifying the authenticity of public keys and for transferring trust among users or business partners. The major issue in PKI is how to revoke a certificate before its expiration. It has been argued that the running expenses of a PKI derive mainly from administering revocation [14]. Various mechanisms have been designed to achieve efficient, timely, and scalable revocation of certifications [15, 6].

The certificate revocation list (CRL) mechanism was introduced in 1988 and since then it remains the most common and simplest method for certificate revocation. A CRL is a time-stamped list of certificates which have been revoked before their expiration. A CA issues a signed CRL periodically so as to maintain a good synchronization between certificate users and revocation source. Some extensions of CRL include delta-CRL (which only carries changes from previous CRL), partitioned CRL (which is partitioned into a family of CRLs), and indirect-CRL (which can be issued by different CA than issuer of certificates). Rivest proposed to use short-lived certificates so as to eliminate CRLs [13]. The major drawbacks of this approach include a high burden placed on certificate servers which need to sign more certificates, as well as the problem of key compromise which cannot be addressed without using a separate mechanism [9].

Micali introduced the certificate revocation system (CRS) which is different from CRL. In CRS, a CA signs a fresh list of all not-yet-expired certificates together with selected hash chain values. A user sending a request regarding the validity of a single certificate will get a response including two hash chain values. The hash chain values can be used to verify whether the queried certificate is valid or not for a certain time interval. The major advantage of this method is that the verification process is very efficient, thus can be performed on-line. However, as pointed out by Naor and Nissim [11], the main disadvantage of this system is the increase of the CA's communication cost.

The certificate revocation tree (CRT) mechanism was suggested by Kocher [7] which can be used by a verifier of a certificate to obtain a short proof if the certificate has not been revoked. A CRT is a hash tree whose leaf nodes correspond to a set of statements about certificates status. The set of statements provides information about whether a certificate is revoked or not. A proof for a certificate status consists of an appropriate path in the hash tree (from the root to a leaf) specifying for each node the values of its children. With CRT, a user may hold a short proof for the validity of his certificate such that the entire CRL is not necessary for verifying the status of the certificate. The drawback of CRT is its maintenance cost. Any change to the set of revoked certificates may cause re-computation of the entire CRT.

An alternative to the CRL mechanism is to use on-line certificate status protocol (OCSP) to reduce the latency between a revocation report and the distribution of revocation information to users [10]. Once a CA accepts a revocation report, any query (OCSP request) to the status of one or more certificates will be correctly answered by an on-line validation server (OCSP responder) with relevant status values (good, revoked, or unknown) and valid intervals. Though OCSP provides more timely revocation services, it imposes new security requirements as the certificate validators shall trust the on-line validation service.

Besides the above mechanisms, researchers have studied various aspects of certificate revocations including the meaning of revocation [3, 4], the model of revocation [2], communication cost of revocation [11], tradeoffs in certificate revocation schemes [16], and risk management in certificate revocation [8]. Though various tradeoffs have been studied for different revocation options, no attempt has been made to understand the probability distribution of request for certificate revocation. In this paper, we conduct such research for CRL releasing mechanism based on real data, and give concrete guidance for the optimal operation of CA in various scenarios.

3 Problem Formulation

In this paper, we study how often should a CA release its CRLs. There are several key assumptions in our study: 1) This is a monopoly case, which means either there is just one CA in the system or different CAs provide different types of certificate services. So CAs do not need to consider the competition effect. 2) CA already decides the issued age of a given type of certificates, where issued age is defined as the time difference between the expired date and the issued date. 3) To get started, we assume CA issues one type of certificates with the same issued age. These certificates are independent and identical in terms of risk and cost. Later we will move on to more general cases.

Given all of the assumptions, the goal of CA is to find out about how often it should release a CRL to minimize its operational cost over a given period. Here we define "how often" as the optimal time interval between two successive CRLs being released, and the "operational cost" as the sum of *variable cost, fixed cost*, and *liability cost* as defined below.

Normally CA takes a batch process for CRL release. There is a trade-off between cost and risk. In the case that consumer files a revocation request to CA but CA does not release a CRL on time, we assume that CA will bear the liability cost if there is any damage occurred between request filing and CRL releasing. Each time CA releases a CRL, it incurs both fixed cost component and variable cost component. The fixed cost does not change with the length of CRL. It indicates a fixed dollar amount each time CA spends for releasing one CRL, regardless of the number of certificates in that CRL. Variable cost is the cost associated with processing each individual certificate revocation request.

If CA releases the CRL too often, its liability cost is low, but its fixed cost and variable cost will be high. On the other hand, the saving on fixed cost and variable cost might not be offset by the increasing liability cost if CA releases the CRL too rarely. So CA needs to find an optimal interval for CRL release. In order to find this solution, CA must know the CRL length on a given day if it decides to publish CRL on that day. The length of CRL at any time t is related to three components: 1) Length of CRL at any time $t-1$. 2) How many revoked certificates including in the CRL at time $t-1$ will be expired at time t. According to CRL policy, if the certificate is expired, it should be excluded from CRL. 3) How many new revocation requests it will receive from time $t-1$ to t.

Parameter	Meaning of Parameter
a	Max number of days between two successive CRL released dates that is accepted by customers.
b	The average percentage of certificates revoked among that type of certificates issued.
c	The number of days between two CRL releasing dates.
t	Time parameter in the function of R(t),which is R(t)=ke^{-kt}.
Δt	Time interval between two generations of CRLs.
X	Date on which certificates get issued.
k	Parameter in the function of R(t),which is R(t) = ke^{-kt}.
n	Numbers of generations.
v	Any time between 0 and ∞ in the f(v),F(v), and P(v).
d	The upper bound of the number of certificate revocations in one CRL which is allowed by CA, before it releases the CRL.
q	The number of CRLs that CA will published during period β .
i	The ith CRL published by CA.
Nd_i	CA releases the CRL on the Nd_i day.
α	Number of certificates issued at different times.
β	Issued Age of CRL, which is equal to Expired Date Minus Issued Date.
μ	Shape parameter in Poisson distribution which indicates the average number of certificate revocations in a given time interval.
λ	Number of certificate revocations in CRL on the day β for Poisson case.
θ	Stable number of certificates in CRL on a given day after β if CA decides to release CRL on that day.
FC	The fixed cost of CA for publishing one CRL.
VC	The constant unit cost of CA for including one certificate into the CRL.
Υ	The expected risk/liability per revocation cost for CA for delaying publish that revocation for one day
$f(v)$	The number of new certificate revocations between day v and day v+Δt.
$F(v)$	The valid cumulative number of certificate revocations from time 1 to v.
$P(v)$	The percentage of certificate revocations occurred from time v to time v+Δ t.

Table 1: Notation

For ease of reference, Table 1 lists all the notation that will be used in this paper.

4 Data Collection

How many new revocation requests a CA received from time $t-1$ to t is really driven by the probability distribution of certificate revocation requests. From September 7th to September 13th , we collected a series of CRLs from VeriSign.com. As one of the biggest Certificate Authority in the world, VeriSign provides variant types of certificates and publishes different CRLs periodically. We randomly choose five different CRL files from

VeriSign website, which belong to five different classes. Table 2 provides the descriptions for these 5 CRL files which have 39,243 total revocation records. When a certificate is issued, its validity is limited by an expiration date. Note that the definition of issued age is:

$$\text{Issued Age} = \text{Expired Date} - \text{Issued Date}$$

However, there are circumstances where a certificate must be revoked prior to its expiration date. Thus, the truly existence age of the certificate is the time between the issued date and the revoked date.

$$\text{Existence Age} = \text{Revoked Date} - \text{Issued Date}$$

A certificate is valid for its issued age unless it is revoked. Each revoked certificate in a CRL is identified by its certificate serial number and the revoked date. Based on the serial number of a given certificate, we searched the VeriSign online database to get both the issued date and the expired date correspondingly. We cleaned those error records whose revoked date was later than the expired date or whose issued date was later than the revoked date.

5 Data Analysis

We present the summary statistics for our data in Table 3. The average issued age of these CRLs is 493 days, while the average existence age is much shorter, only 31 days. To further demonstrate what is happening here, we plot the number of revocations against existence age in Figure 1, and the percentage of revocations against existence age in Figure 2 for classes RSASecureServer and SVRIntl.

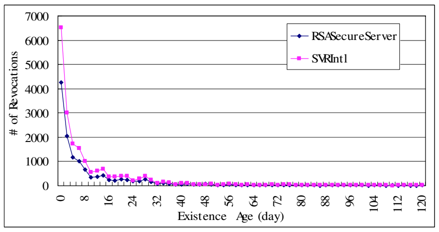


Figure 1: Number of revocations vs. existence age

The most interesting finding is that most of the certificate revocations occur at the first few days after issued, and the percentage of revocations decreases with elapsed time. More than 30% of revocations occur within the first two days after certificates get issued. This distribution pattern is very robust, and it is insensitive to which CRLs we investigated and which years we selected. It still holds when we pool five CRLs together.

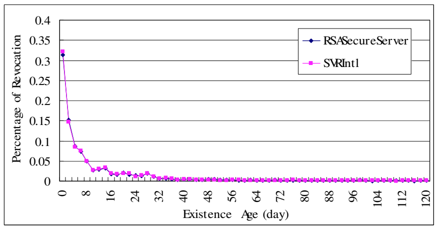


Figure 2: Probability of revocation vs. existence age

5.1 Empirical Model

In order to get the size of CRL at any time t, we must know the behavior of the probability density function (PDF) of certificate revocations over time. In the above, empirically we already show that the percentage of revocations decreases with elapsed time. Next statistically we derive the underline PDF.

5.1.1 Underline PDF for Certificates Issued at a Particular Time

We assume that there are α certificates issued at time X with issued age β. To get start, we assume that α is a constant number. Later we change α to a random number with a Poisson distribution to study the more general case. From time X to time $X+\beta$, on average $\alpha b\%$ of the certificates will be revoked. At time $X+\beta$, all the certificates issued at time X will be expired, no matter whether they have been revoked or not. Let $R(t)$ be the probability that any given certificate issued at time X will be revoked in the interval $[t, t+\Delta t]$, where t is between X and $X+\beta$. It also represents the revoked percentage, which is the number of revocations occurred in the interval $[t, t+\Delta t]$ divided by the total number of revocations occurred between X and $X+\beta$ (i.e., $\alpha b\%$). Following the empirical distribution observed in Figure 2, we use an exponential probability density function to model this distribution.

$$R(t) = ke^{-kt} \tag{1}$$

We use Maximum Absolute Deviation (MAD) to determine the parameter k. The MAD is proposed by Kolmogorov-Smirnov. It minimizes the largest gap between the cumulative relative frequency of a given data set and that of its fitted statistical distribution. In Figure 3, we present both the real empirical data and theoretically fitted PDF[4]. The PDF fits the empirical data very well when the parameter k is equal to 0.26, which is accepted at a 99% confidence interval.

File Name	Issuer	Publishing Time	Purpose	No.of Items	Max Existence Age (day)	Signature Algorithm
Class3CodeSigning2001.crl	VeriSign Class 3 Code Signing 2001 CA	September 04, 2005 6:00:08 PM	Code signing and object signing certificates used for Netscape browsers, Microsoft Internet Explorer browsers, Microsoft Office, Sun Java Signing, Macromedia, and Marimba.	1,993	380	md5RSA
CSC3-2004.crl	VeriSign Class 3 Code Signing 2004 CA	September 11, 2005 6:00:25 PM	Same as VeriSign Class 3 Code Signing 2001 except used for certificates with different expiration date.	228	364	md5RSA
Class3 NewOFX.cr	VeriSign Class 3 Open Financial Exchange CA	September 11, 2005 6:00:15 PM	Open financial exchange certificates, used for authenticating and securing commerce on the Internet.	515	302	md5RSA
RSASecure Server.crl	RSA Secure Server CA	September 08, 2005 6:00:25 PM	Secure server certificates used by a Root CA for managing PKI for SSL Customers and VTN Affiliates.	14,837	727	md5RSA
SVRIntl.crl	VeriSign International Server CA Class 3	September 08, 2005 6:00:16 PM	Global server certificates for managing PKI for SSL (Premium Edition) customers and VTN Affiliates.	21,839	720	md5RSA

Table 2: Descriptions of CRL files

Items	Issued Age(Unit:Day)	Existence Age(Unit:Day)
Mean	493	31
Median	366	5
Max	1173	727
Min	1	0
Q3	730	21
Q1	365	1

Table 3: Summary statistics

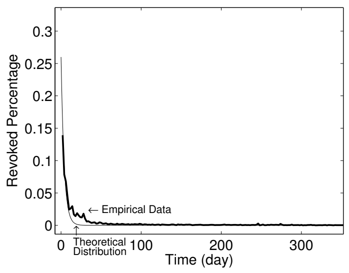


Figure 3: Empirical data vs. fitted exponential PDF

5.1.2 First Case: A Model for Pooling Certificates Issued at Different Time

In the previous section, we only consider the PDF of one population of certificates issued at time X. Now consider that CA issues certificates at different time. Each generation of certificates is composed of certificates issued at a particular time with the same issued age β, where the time interval between two successive generations is Δt. At any given time interval $[t, t + \Delta t]$, the revocation requests CA received originate from different generations.

In order for CA to decide when to release the CRLs, on daily basis CA must know: 1) The number of new revocation requests; and 2) The size of the CRL if it decides to release the CRL on that day. Based on the PDF derived from the previous section, we build a model to compute the total number of certificate revocations when pooling revocations from different generations.

The number of new revocation requests The PDFs of revocations from different generations of certificates follow the same exponential probability density $R(t) = ke^{-kt}$ as shown in Figure 4. Suppose that v is any time in $(0, \beta]$. Let $f(v)$ be the number of new certificate revocations between day v and day $v + \Delta t$, from all of the valid generations.

$$\begin{aligned} f(v) &= \alpha b\%R(v) + \alpha b\%R(v-\Delta t) + \\ &\quad \alpha b\%R(v-2\Delta t) + \ldots + \\ &\quad \alpha b\%R[v-(n-1)\Delta t] \end{aligned} \tag{2}$$

where n is the number of generations in time period β; that is, $n = \lceil \frac{\beta}{\Delta t} \rceil$.

Assuming that Δt is one day, and that v is an integer, where v is in $(0, \beta]$, then we get the following equation.

$$\begin{aligned} f(v) &= \alpha b\%R(1) + \alpha b\%R(2) + \ldots + \alpha b\%R(v) \\ &= \alpha b\%ke^{-k} + \alpha b\%ke^{-2k} + \ldots + \alpha b\%ke^{-vk} \\ &= \alpha b\%ke^{-k}\frac{1-e^{-vk}}{1-e^{-k}} \end{aligned} \tag{3}$$

When v is in (β, ∞), we have

$$\begin{aligned} f(v) &= \alpha b\%R(1) + \alpha b\%R(2) + \ldots + \alpha b\%R(\beta) \\ &= \alpha b\%ke^{-k} + \alpha b\%ke^{-2k} + \ldots + \alpha b\%ke^{-\beta k} \\ &= \alpha b\%ke^{-k}\frac{1-e^{-\beta k}}{1-e^{-k}} \end{aligned} \tag{4}$$

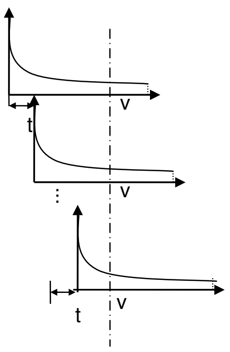


Figure 4: Model for pooling revocations from different generations

Equations 3 and 4 show that the number of new revocation requests CA received on daily basis increases with a decreasing rate as time elapses from day zero until day β. After that, the number of revocation requests on daily basis becomes a constant number. Figure 5 shows the number of new certificate revocations on daily basis in $(0, 2\beta]$ for the case where $\alpha = 1000$, $b\% = 10\%$, $k = 0.26$, and $\beta = 36$ days. From now on we omit the graph for $(m\beta, (m+1)\beta]$, where $m \geq 2$, because the shape in those regions are the same as that in $(\beta, 2\beta]$.

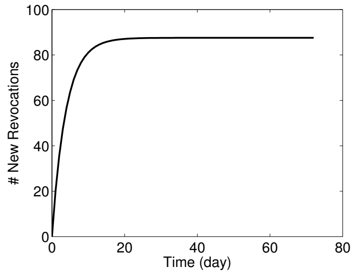


Figure 5: $f(v)$ behavior: the number of new certificate revocations on daily basis

on day v, includes the new revocation requests on day v as well as the valid historical revocation requests (occurred before day v) whose expiration day is later than v. Let $F(v)$ be the valid cumulative number of certificate revocations from time 1 to v, where "valid" means not expired. This is also the size for the CRL if CA decides to publish it on that day. For any time $v \in (0, \beta]$, we have

$$\begin{aligned} F(v) &= \sum_{t=1}^{v} f(t) = \sum_{t=1}^{v} \alpha b\% k e^{-k} \frac{1-e^{-tk}}{1-e^{-k}} \\ &= \frac{\alpha b\% k e^{-k}}{1-e^{-k}} [v - \frac{e^{-k}}{1-e^{-k}} (1 - e^{-vk})] \quad (5) \end{aligned}$$

Considering the $(\beta + 1)$ day, the revoked certificates from the first generation are expired, and thus removed from the CRL. At this time, the number of valid generations is $(\beta + 1) - 1 = \beta$.

Considering the $(\beta + 2)$ day, revocation from the first two generations are expired, and thus removed from the CRL. At that time the number of valid generations is also $(\beta + 2) - 2 = \beta$.

The rest may be deduced similarly. For any $v \in (\beta, +\infty)$, we have

$$\begin{aligned} F(v) &= F(\beta) = \sum_{t=1}^{\beta} f(t) \\ &= \frac{\alpha b\% k e^{-k}}{1-e^{-k}} [\beta - \frac{e^{-k}}{1-e^{-k}} (1 - e^{-\beta k})] \quad (6) \end{aligned}$$

Equations 5 and 6 show that the size of CRL on daily basis with respect to time is a convex function. It increases with an increasing rate as time elapses from day 0 until the time reaches issued age β. After that, the size of CRL becomes a constant number. The reason it is not going to be infinite is that at any time some revoked certificates may be expired and removed from CRL. Figure 6 shows the daily CRL size in $(0, 2\beta]$ for the case of $\alpha = 1000$, $b\% = 10\%$, $k = 0.26$, and $\beta = 36$ days.

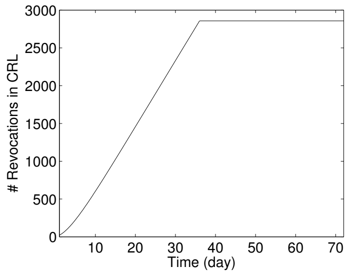


Figure 6: $F(v)$ behavior: daily size of CRL

$P(v)$ be the percentage of certificate revocations occurred from time v to time $v + \Delta t$, which is defined as the number of new certificate revocations at time v divided by the cumulative valid number of certificate revocations from time 1 to v. For any time $v \in (0, \beta]$, we have

$$\begin{aligned} P(v) &= f(v) / \sum_{t=1}^{v} f(t) \\ &= \frac{\alpha b\% k e^{-k} \frac{1-e^{-vk}}{1-e^{-k}}}{\frac{\alpha b\% k e^{-k}}{1-e^{-k}} [v - \frac{e^{-k}}{1-e^{-k}} (1 - e^{-vk})]} \\ &= \frac{1}{\frac{v}{1-e^{-vk}} - \frac{e^{-k}}{1-e^{-k}}} \quad (7) \end{aligned}$$

For any time $v \in (\beta, +\infty)$, the number of new certificate revocations at time v is always equal to $f(\beta)$, and the cumulative valid number of certificate revocations from time 1 to v is always equal to $F(\beta)$. Thus,

$$\begin{aligned} P(v) &= f(\beta)/\sum_{t=1}^{\beta} f(t) \\ &= \frac{\alpha b\% k e^{-k} \frac{1-e^{-\beta k}}{1-e^{-k}}}{\frac{\alpha b\% k e^{-k}}{1-e^{-k}}[\beta - \frac{e^{-k}}{1-e^{-k}}(1-e^{-\beta k})]} \\ &= \frac{1}{\frac{\beta}{1-e^{-\beta k}} - \frac{e^{-k}}{1-e^{-k}}} \end{aligned} \tag{8}$$

Equations 7 and 8 show that the majority of the certificate revocations occur at early stage of the issued age of certificates, right after they were issued. Figure 7 shows the graph of percentage of certificate revocations in $(0, 2\beta]$ for the case of $\alpha = 1000$, $b\% = 10\%$, $k = 0.26$, and $\beta = 36$ days.

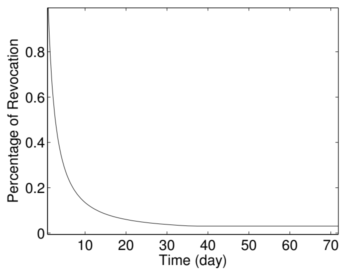


Figure 7: $P(v)$ behavior: percentage of revocations

5.1.3 Second Case: Overlap of Certificates with different Issued Ages

In the first case, we assume that CA issues a fixed number of certificates at different time, but each time it is the same type of certificates that issued with the same issued age. Now we relax these assumptions to a more general case by assuming that at any point of time CA can issue two types of certificates with different issued ages β_1 and β_2, where $\beta_2 > \beta_1$. We assume that these two types are independent of each other. Under these assumptions, we compute the new $F(v)$ and $P(v)$ in different intervals of $(0, \beta_1]$, $(\beta_1, \beta_2]$, and $(\beta_2, +\infty)$ correspondingly.

We overlap two types of certificates with the distribution functions $R_1(t) = k_1 e^{-k_1 t}$ and $R_2(t) = k_2 e^{-k_2 t}$. At the same time, each type of certificates is composed of generations of different certificates issued at different time.

the daily CRL size $F(v)$ is a cumulative number of the revocations of two types of certificates. For any time $v \in (0, \beta_1]$, we have

$$F(v) = \sum_{t=1}^{v} f_1(t) + \sum_{t=1}^{v} f_2(t) \tag{9}$$

In the interval $(\beta_1, \beta_2]$, the size of CRL for the certificates whose issued age is β_1 has become stable and the value of $F_1(v)$ will be constant, while that for the certificates whose issued age is β_2 keeps increasing. For any $v \in (\beta_1, \beta_2]$, we have

$$F(v) = F_1(\beta_1) + \sum_{t=1}^{v} f_2(t) \tag{10}$$

In the interval $(\beta_2, +\infty)$, both CRLs become stable, and thus

$$F(v) = F_1(\beta_1) + F_2(\beta_2) \tag{11}$$

Figure 8 shows the graph of the cumulative numbers of valid certificate revocations in $(0, +\infty)$ for the case $k_1 = 0.26$, $\beta_1 = 36$, $k_2 = 1$, $\beta_2 = 72$, $\alpha_1 = 2000$, $\alpha_2 = 1000$, $b_1 = 10\%$, and $b_2 = 10\%$.

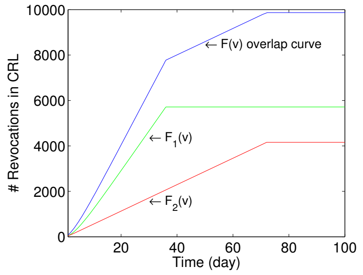


Figure 8: $F(v)$ behavior: overlap certificates with different issued ages

$P(v)$ be the percentage of certificate revocations occurred between v and $v + \Delta t$ for the pooling case. For any $v \in (0, \beta_1]$, we have

$$P(v) = \frac{f_1(v) + f_2(v)}{\sum_{t=1}^{v} f_1(t) + \sum_{t=1}^{v} f_2(t)} \tag{12}$$

For any $v \in (\beta_1, \beta_2]$, we have

$$P(v) = \frac{f_1(\beta_1) + f_2(v)}{\sum_{t=1}^{\beta_1} f_1(t) + \sum_{t=1}^{v} f_2(t)} \tag{13}$$

For any $v \in (\beta_2, +\infty)$, we have

$$P(v) = \frac{f_1(\beta_1) + f_2(\beta_2)}{\sum_{t=1}^{\beta_1} f_1(t) + \sum_{t=1}^{\beta_2} f_2(t)} \tag{14}$$

Figure 9 shows the graph of percentage of revocations in interval $(0, +\infty)$ for the case $k_1 = 0.26$, $\beta_1 = 36$, $k_2 = 1$, $\beta_2 = 72$, $\alpha_1 = 2000$, $\alpha_2 = 1000$, $b_1 = 10\%$ and $b_2 = 10\%$. Overall, the behaviors of $F(v)$ and $P(v)$ are almost the same as those in case 1.

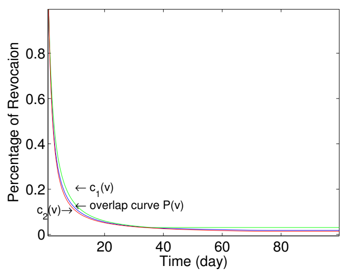


Figure 9: $P(v)$ behavior: overlap certificates with different issued ages

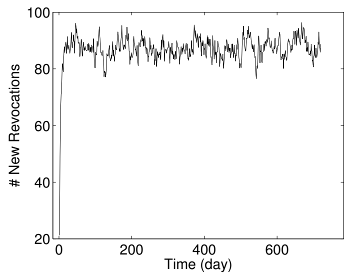


Figure 10: $f(v)$: daily number of new certificate revocations with Poisson distribution

5.1.4 Third Case: Simulation in the Case of Poisson Distribution

In our first case, we assume that CA issues a fixed number of certificates at different time. A more general case is that CA issues α certificates, where α is a random number following a Poisson distribution with parameter μ. For this case, the average number of revocation requests per interval is μ. The probability that there are x revocation requests occurred in each interval is $P_\mu(x) = \frac{\mu^x e^{-\mu}}{x!}$. Because the explicit forms of $f(v)$ and $F(v)$ are messy, we uses simulation to get some insights.

We conducted our simulation on a HP 1940 PC (with Pentium 4 CPU and 1.00GB RAM) using Visual C++. We follow the steps below in our simulation:

1. Firstly, generate a sequence of random number α_1, $\alpha_2, \alpha_3, \ldots, \alpha_m$ based on the value of μ.

2. Secondly, compute the total number of new revocations on daily basis between 1 and 2β, which is $f(v)$.

3. Thirdly, compute the valid cumulative number of revoked certificates from day 1 to day 2β, which is $F(v)$.

4. At last, generate different groups of random numbers for α, repeat step 1 to step 4 for twenty times.

Figure 10 shows a typical case for daily numbers of new certificate revocations with Poisson distribution when $\beta = 360$ and $k = 0.26$. The number of new certificate revocations increases sharply to about 90 within a very short period of time after the certificates get issued. Instead of becoming stable as in case 1 and case 2, the curve fluctuates around 90 after a short period of time. The oscillation is driven by the randomness introduced by using Poisson distribution.

Figure 11 shows that $F(v)$ continues increasing from 1 to β, where $\beta = 360$. After β, it begins to fluctuate. In a actual business environment, a typical issued age β is 360 days, which is much bigger than Δt. In such a case, the stable value of λ in Figure 11 is so large that it dominates the fluctuation introduced by Poisson. Consequently, the curve is very smooth after β. This is consistent to the case 1 when we assume a fixed number of certificates issued at any point of time. When CA decides how often CRL should be released, it mostly cares about the distribution of $F(v)$. Because of the existence of the similarity between the fixed number case and the Poisson distribution case for the $F(v)$ distribution, later for our economic analysis, we will focus on the fixed number case.

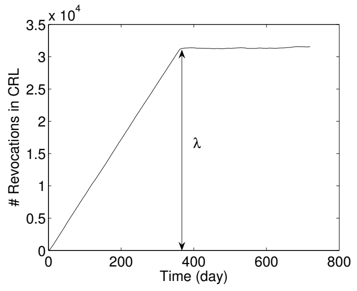


Figure 11: $F(v)$: daily size of CRL with Poisson distribution

5.2 Analytical Model: How Often Should CA Release CRLs

The key research question in our paper is to give a prescription to CA to decide how often it should release its CRLs. In order to answer that question, CA must know the distribution of new certificate revocation $f(v)$ and the distribution of certificate revocation list $F(v)$. CA needs to balance the liability cost of not releasing CRL on time and the fixed and variable costs of releasing CRL too often. So the goal for CA is to minimize the overall operational cost. Because the behaviors of $f(v)$ and $F(v)$ when time t is greater than the issued age β are different from those when time t is smaller than β, we analyze the optimal strategies for CA for these two cases separately. For each case we assume a monopoly case so that there is no competition between CAs, and that the certificates are homogeneous in terms of risk, cost, and revocation probability. Also, different types of certificates are independent from each other. CA will get an optimal CRL releasing strategy for each type of certificates based on properties of the certificates.

5.2.1 Optimal Releasing Strategy When Time is Greater Than β

When time is greater than β, CA has run certificate services for at lease one issued age for that type of certificates. We will use the following variables in our analysis (the numbers in parentheses are the default values used in our computation).

- β: The issued age of one type of CRL. ($\beta = 360$)
- c: The estimated number of days between two CRL releasing dates. This is the decision variable that CA needs to optimize.
- θ: Estimated numbers of certificates in a CRL on a given day after issued age β if CA decides to release CRLs on that day. According to case 1, $\theta = F(\beta) = \frac{\alpha \cdot b\% \cdot k \cdot e^{-k}}{1-e^{-k}}(\beta - \frac{e^{-k}}{1-e^{-k}})$. ($\theta = 32,000$, $k = 0.26$, and $\beta = 360$)
- FC: The fixed cost for CA to publish one CRL .($FC = \$10,000$)
- VC: The variable cost for CA to include each individual certificate into a CRL. We assume the VC does not changed with the length of CRL. ($VC = \$1$)
- Υ: The expected liability cost per certificate revocation if CA delay publishing the revocation for one day. Therefore, the risk of delaying publishing a CRL of θ certificate revocations for n day is $\Upsilon \cdot \theta \cdot n$. If we assume that for the whole period of β, the expected liability cost that CA pays for the accident caused by the delay of publishing CRLs is Qm; (i.e.,$Qm = \$100,000$), then $\Upsilon = Qm/(\theta * \beta) = \$0.0087$ for $\theta = 32,000$ and $\beta = 360$.
- a: Recency requirement set up by the customers. It is the max number of days between two successive CRL releasing dates that is acceptable by customers. ($a = 50 days$)

Because $f(v)$ and $F(v)$ are stable after β, CA can take either a fixed interval strategy or a fixed CRL size strategy for releasing CRL. Fundamentally these two strategies are inter-exchangeable. For simplicity reason, we present the solution for the fixed interval strategy.

If CA releases one CRL every c days, the total cost for CA within period β is

$$cost(c) = [\Upsilon \cdot \theta \cdot \sum_{n=0}^{c-1} n + FC + \theta \cdot VC] \cdot \frac{\beta}{c} \tag{15}$$

converts to the following optimization problem:

$$\begin{cases} \min . \, cost(c) \\ s.t. FC \gg VC > 0, \Upsilon > 0, c \leq a \end{cases} \tag{16}$$

According to Karush-Kuhn-Tucker theorem[5][12], we get

$$\begin{cases} \frac{\partial[cost(c)+L(c-a)]}{\partial c} = 0 \\ c - a \leq 0 \\ L \geq 0 \\ L(c-a) = 0 \end{cases} \tag{17}$$

In order for $L \cdot (c-a) = 0$ to hold, it is required either (i) $L = 0$ or (ii) $c - a = 0$. We consider case (i) and case (ii) in the following.

Case (i) If $L = 0$, then

$$\begin{cases} \frac{\partial[cost(c)+L(c-a)]}{\partial c} = 0 \\ c - a \leq 0 \end{cases} \tag{18}$$

Compute the first derivation of the cost with respect to c, and get the optimal result c_0:

$$\begin{cases} c_0 = \sqrt{(\frac{2}{\Upsilon})(\frac{FC}{\theta} + VC)} \\ c_0 \leq a \end{cases} \tag{19}$$

Compute the second derivation of cost with respect to c, and the result is

$$\frac{\partial^2 cost(c)}{\partial c^2} = \frac{2(FC \cdot \beta + \theta \cdot VC \cdot \beta)}{c^3} \tag{20}$$

Because the second derivation of $cost(c)$ at point c_0 is

$$\frac{\partial^2 cost(c)}{\partial c^2}|_{c=c_0} = \frac{\Upsilon \cdot VC \cdot \beta}{\sqrt{(\frac{2}{\Upsilon})(\frac{FC}{\theta} + VC)}} > 0, \tag{21}$$

$cost(c)$ reaches its minimum value at c_0. The minimum operational cost of CA is

$$\begin{aligned} &[\Upsilon \cdot \theta \cdot \sum_{n=0}^{c_0-1} n + FC + \theta \cdot VC] \cdot \frac{\beta}{c_0} = \\ &\Upsilon \cdot \theta \cdot \beta(c_0 - \frac{1}{2}) = \\ &\Upsilon \cdot \theta \cdot \beta(\sqrt{(\frac{2}{\Upsilon})(\frac{FC}{\theta} + VC)} - \frac{1}{2}) \end{aligned} \tag{22}$$

Case (ii) If $c-a=0$, the function is

$$\begin{cases} L = \frac{FC\cdot\beta+\theta\cdot VC\cdot\beta}{\Delta t^2} - \frac{\Upsilon\cdot\theta\cdot\beta}{2} \geq 0 \\ c=a \end{cases} \tag{23}$$

The minimal cost of CA is fixed and equal to

$$[\Upsilon\cdot\theta\cdot\sum_{n=0}^{a-1} n + FC + \theta\cdot VC]\cdot\frac{\beta}{a} = \frac{1}{2}\cdot\Upsilon\cdot\theta\cdot\beta\cdot(a-1)+\frac{\beta}{a}(FC+\theta\cdot VC) \tag{24}$$

when

$$a \leq \sqrt{(\frac{2}{\Upsilon})(\frac{FC}{\theta}+VC)} \tag{25}$$

We can see that the minimum releasing interval CA should follow is either a or $c_0 = \sqrt{(\frac{2}{\Upsilon})(\frac{FC}{\theta}+VC)}$ depending on whether $\sqrt{(\frac{2}{\Upsilon})(\frac{FC}{\theta}+VC)}$ is greater than a or not. It is clear that $\sqrt{(\frac{2}{\Upsilon})(\frac{FC}{\theta}+VC)}$ is an increasing function of FC and VC, but a decreasing function of Υ and β, where $\theta = F(\beta) = \frac{\alpha b\%ke^{-k}}{1-e^{-k}}[\beta - \frac{e^{-k}}{1-e^{-k}}(1-e^{-\beta k})]$. That means if the fixed cost or the variable cost is higher, or the liability cost is lower, or the issued age of the certificates is shorter, CA should release CRLs less frequently.

example to demonstrate how much money CA can save by following our strategy. The best waiting days to achieve the minimal cost is $c = \sqrt{(\frac{2}{\Upsilon})(\frac{FC}{\theta}+VC)} = 17.37 \leq a = 50$ days. Figure 12 shows the total cost for CA by using different releasing strategies. If CA deviates from the $c_0 = 17$days by using $2\cdot c_0 = 34$days, CA ends up spending almost $\$400,000$ more for just one type of certificates within a period of β. This is not a trivial number given that there are multiple CAs providing numerous certificate services.

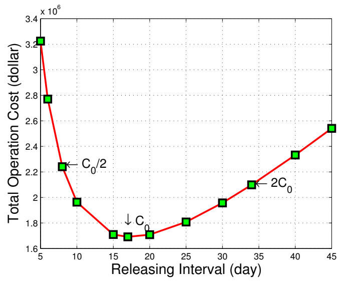


Figure 12: Total cost of CA with different releasing interval after β

5.2.2 Optimal Releasing Strategy When Time is Smaller Than β

There are two possible business scenarios for this case: 1) A "grown-up" CA that has been in CRL business for quite a while, but faces the situation of providing CRL services for a new type of certificates. 2) A "start-up" CA that just begin to provide CRL services. For these two cases, v is inside $(0,\beta]$, and $F(v)$ is a convex function with respective to v. CA can take either a fixed interval or a fixed size strategy for releasing CRLs. The fixed size means that a CA will release the CRLs whenever the number of certificates included in the CRL exceeds a fixed pre-specified number. Next we analyze both cases by using simulation. For each case, CA can get the parameter estimators either based on other types of certificates it provides before or from its industry peers.

the analysis given for the case when time is greater than β, we can obtain the cost function when the time is smaller than β:

$$\begin{aligned} &cost(c) = \\ &\frac{\Upsilon ab\%ke^{-k}}{1-e^{-k}}\sum_{x=0}^{\frac{\beta}{c}-1}\sum_{n=0}^{c-1}(c-n)(1-e^{-(xc+n+1)k}) + \\ &\frac{\beta}{c}FC + F(\beta)VC = \\ &\frac{\Upsilon ab\%ke^{-k}}{1-e^{-k}}(\beta c - \frac{\beta(c-1)}{2} + \\ &\frac{((c+1)e^{-2k}-ce^{-k}+e^{-(c+2)k})(1-e^{-\beta k})}{(1-e^{-k})^2(1-e^{-ck})}) + \\ &\frac{\beta}{c}FC + \\ &\frac{\alpha b\%ke^{-k}}{1-e^{-k}}[\beta - \frac{e^{-k}}{1-e^{-k}}(1-e^{-\beta k})]VC \end{aligned} \tag{26}$$

For each possible c, ranging from 1 to β, we compute the total cost for CA. Figure 13 shows[6] the total cost for CA by using different releasing intervals. We find that when $c = 28$ days, we get the minimal cost $\$2.08603 * 10^5$. This means that when time is smaller than β, CA should release CRLs once every 28 days. Recall that the optimal interval is 17 days when time is greater than β. It is easy to know that $cost(17) = \$2.64014 * 10^5 > cost(28) = \$2.08603 * 10^5$ in the period of $(0,\beta]$; therefore, CA should take different strategies for time periods $(0,\beta]$ and $(\beta,+\infty)$.

following variables for analyzing the fixed size strategy:

- d: CA will publish a new CRL if the number of certificate revocations exceeds d.
- q: Estimated numbers of CRLs that CA will publish during one issued age from time 0 to β.

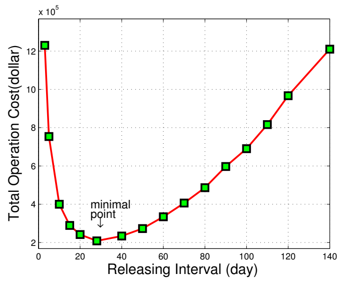


Figure 13: Total cost of CA with different releasing interval before β

- i: The i-th CRL published by CA, where $0 < i \leq q$.
- Nd_i: CA releases the CRL on the Nd_i-th day.
- $F(v)$: Size of CRL at time v.

Then we have

$$F(v) = \frac{\alpha b\% k e^{-k}}{1-e^{-k}}[v - \frac{e^{-k}}{1-e^{-k}}(1-e^{-vk})] \qquad (27)$$

In order to estimate Nd_i, we need to compute the inverse function of $F(v)$, which is denoted as $G(d)$ (i.e., $G(d) = F^{-1}(v)$. After that, we can get the exact day Nd_i, on which CA needs to release its CRLs, by solving $Nd_1 = G(d)$, $Nd_2 = G(2*d), \ldots, Nd_i = G(i*d)$.

Given the LanbertW function defined as

$$LambertW(x) * exp(LambertW(x)) = x \qquad (28)$$

we have

$$\begin{aligned} G(d) &= LambertW \cdot d(e^k - 1) \\ &\cdot \frac{e^{(-\frac{d(e^k)^2 - 2de^k + d + 100k}{100e^k})}}{100e^k k} \\ &+ \frac{de^{2k} - 2de^k + d + 100k}{100e^k k} \end{aligned} \qquad (29)$$

We conduct our simulation step by step. Firstly, for each possible d chosen from 100 to $36,000$ ($a = 1000$, $b\% = 10\%$), compute $Nd_1, Nd_2, \ldots, Nd_i$. Secondly, calculate the time difference between Nd_i and $Nd_i - 1$, which we call c_i, to estimate the liability cost. Thirdly, compute $cost(d)$ for each individual d as following

$$\begin{aligned} cost(d) &= \frac{\Upsilon \cdot a \cdot b\% \cdot k \cdot e^{-k}}{1-e^{-k}} \\ &\times \sum_{j=0}^{q-1}\sum_{x=0}^{c_i - 1}(c_i - x)(1 - e^{-((i-1)c_i + x + 1)k}) \\ &+ q \cdot FC + F(\beta) \cdot VC \end{aligned} \qquad (30)$$

We find that the minimal cost is $\$2.26790 * 10^5$ when $d = 2800$. The minimal cost is very similar to the result that we obtained using the optimal fixed interval. Therefore, when time is smaller than β, CA can take either a fixed size strategy or a fixed interval strategy. But CA can no longer follow the same optimal releasing interval as in the case when time is bigger than β.

Figure 14 shows the total cost for CA to use different size strategies. The cost for CA is minimal when $d = 2800$.

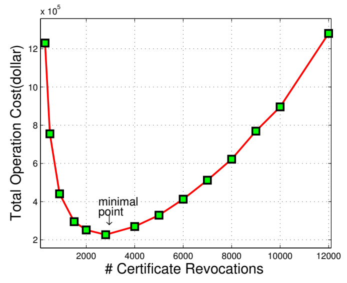


Figure 14: Total cost of CA with different size strategy before β

Figure 15 shows the relationship between releasing time and cumulative revocations when $q = 100$ and $k = 0.26$ for the fixed size strategy. Here we assume that $d = 2800$. That means whenever the size of CRL reaches 2800, CA will release it. That is the reason we see 2800, 5600, 8400, and so on along x-axils. As we can tell, as time moves away from time 0, the releasing interval between two successive CRL releasing dates remains almost unchanged. The fixed size strategy is almost equivalent to the fixed interval strategy at their respectively optimal points.

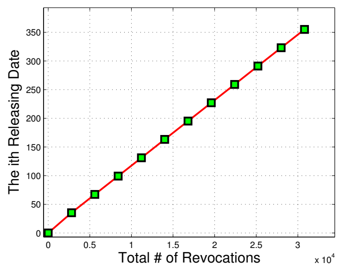


Figure 15: The i-th releasing day vs. cumulative revocations

To summarize, different types of CAs should take different CRL releasing strategies for the same type of certificate services, and the same CA should also use different mechanisms for different types of certificate services.

6 Discussions and Conclusions

In this paper, we analyze real empirical data collected from VeriSign to derive probability density function of certificate revocations. Unlike most previous research, our work is conducted based on real data. The contributions of this paper include: 1) We prove that a revocation system will become stable after a period of time; 2) CA should take different strategies when providing certificate services for a new type of certificates versus a re-serving type of certificates; 3) A start-up CA and a grown-up CA should take different strategies for CRL release; 4) We give the exact steps by which a CA can derive optimal CRL releasing strategies; and 5) We prove that a CA should release CRLs less frequently in the case that the fixed cost is higher, the variable cost is higher, the liability cost is lower, or the issued age of certificates is shorter.

There are several limitations for this study. First, this paper takes a static approach by assuming that there is no correlation between different types of certificates, and that customer behaviors do not affect CA's releasing strategy for deriving the optimal CRL releasing intervals. A more realistic approach is to use game theory to model the interactions between CAs and customers. Second, this paper assumes that CA offers certificates with a fixed issued age. To further minimize the total operational cost, CA may optimize not only the releasing time interval but also the issued age simultaneously.

References

[1] ARNES, A. Public key certificate revocation schemes. Master's thesis, Norwegian University of Science and Technology, 2000.

[2] COOPER, D. A. A model of certificate revocation. In *ACSAC '99: Proceedings of the 15th Annual Computer Security Applications Conference* (Washington, DC, USA, 1999), IEEE Computer Society, p. 256.

[3] FOX, AND LAMACCHIA. Certificate revocation: Mechanics and meaning. In *FC: International Conference on Financial Cryptography* (1998), LNCS, Springer-Verlag.

[4] GUNTER, C. A., AND JIM, T. Generalized certificate revocation. In *Symposium on Principles of Programming Languages* (2000), pp. 316–329.

[5] HOUSLEY, R., FORD, W., POLK, W., AND SOLO, D. RFC 2459: Internet X.509 public key infrastructure certificate and CRL profile, Jan. 1999. Status: PROPOSED STANDARD.

[6] JAIN, G. Certificate revocation: A survey. http://citeseer.ist.psu.edu/511985.html.

[7] KOCHER, P. C. On certificate revocation and validation. In *FC '98: Proceedings of the Second International Conference on Financial Cryptography* (London, UK, 1998), Springer-Verlag, pp. 172–177.

[8] LI, N., AND FEIGENBAUM, J. Nonmonotonicity, user interfaces, and risk assessment in certificate revocation (position paper). In *Proceedings of the 5th Internation Conference on Financial Cryptography (FC'01)* (2002), no. 2339 in LNCS, Springer, pp. 166–177.

[9] MCDANIEL, P., AND RUBIN, A. A response to "can we eliminate certificate revocation lists?". *Lecture Notes in Computer Science 1962* (2001), 245+.

[10] MYERS, M., ANKNEY, R., MALPANI, A., GALPERIN, S., AND ADAMS, C. X.509 internet public-key infrastructure — online certificate status protocol (OCSP). Internet proposed standard RFC 2560, June 1999.

[11] NAOR, M., AND NISSIM, K. Certificate revocation and certificate update. In *Proceedings 7th USENIX Security Symposium (San Antonio, Texas)* (Jan 1998).

[12] POLAK, E. Computational methods in optimization.

[13] RIVEST, R. L. Can we eliminate certificate revocations lists? In *Financial Cryptography* (1998), pp. 178–183.

[14] STUBBLEBINE, S. Recent-secure authentication: Enforcing revocation in distributed systems. In *Proceedings 1995 IEEE Symposium on Research in Security and Privacy* (May 1995), pp. 224–234.

[15] WOHLMACHER, P. Digital certificates: a survey of revocation methods. In *MULTIMEDIA '00: Proceedings of the 2000 ACM workshops on Multimedia* (New York, NY, USA, 2000), ACM Press, pp. 111–114.

[16] ZHENG, P. Tradeoffs in certificate revocation schemes. *Computer Communication Review 33*, 2 (2003), 103–112.

Notes

[1]This work was conducted when Chengyu Ma visited Singapore Management University

[2]http://sign.nca.or.kr/english/english.html

[3]http://www.mozilla.or.kr/zine/?cat=10

[4]We delete those records whose existence ages are zero.

[5]Karush-Kuhn-Tucker condition is a necessary and sufficient optimality condition for constrained optimization problems.

[6]For demonstration purpose, we assume that a is large enough so that CA can adopt a fixed interval determined by any optimal value of our model.

Biometric Authentication Revisited: Understanding the Impact of Wolves in Sheep's Clothing

Lucas Ballard
Department of Computer Science
Johns Hopkins University

Fabian Monrose
Department of Computer Science
Johns Hopkins University

Daniel Lopresti
Department of Computer Science & Engineering
Lehigh University

Abstract

Biometric security is a topic of rapidly growing importance, especially as it applies to user authentication and key generation. In this paper, we describe our initial steps towards developing evaluation methodologies for behavioral biometrics that take into account threat models which have largely been ignored. We argue that the pervasive assumption that forgers are minimally motivated (or, even worse, naïve), or that attacks can only be mounted through manual effort, is too optimistic and even dangerous. To illustrate our point, we analyze a handwriting-based key-generation system and show that the standard approach of evaluation significantly overestimates its security. Additionally, to overcome current labor-intensive hurdles in performing more accurate assessments of system security, we present a *generative attack* model based on concatenative synthesis that can provide a rapid indication of the security afforded by the system. We show that our generative attacks match or exceed the effectiveness of forgeries rendered by the skilled humans we have encountered.

1 Introduction

The security of many systems relies on obtaining human input that is assumed to not be readily reproducible by an attacker. Passwords are the most common example, though the assumption that these are not reproducible is sensitive to the number of guesses that an attacker is allowed. In *online* attacks, the adversary must submit each request to a nonbypassable reference monitor (e.g., a login prompt) that accepts or declines the password and permits a limited number of incorrect attempts. In contrast, an *offline* attack permits the attacker to make a number of guesses at the password that is limited only by the resources available to the attacker, i.e., time and memory. When passwords are used to derive cryptographic keys, they are susceptible to offline attacks.

An alternative form of user input that is intended to be difficult for attackers to reproduce are biometrics. Like passwords, biometrics have typically been used as a technique for a user to authenticate herself to a reference monitor that can become unresponsive after a certain number of failed attempts. However, biometrics also have been explored as a means for generating user-specific cryptographic keys (see for example, [30, 21]). As with password-generated keys, there is insufficient evidence that keys generated from biometric features alone will typically survive offline attacks. As such, an alternative that we and others have previously explored is *password hardening* whereby a cryptographic key is generated from both a password and dynamic biometric features of the user while entering it [22, 23].

While these directions may indeed allow for the use of biometrics in a host of applications, we believe the manner in which biometric systems have been tested in the literature (including our prior work) raises some concerns. In particular, this work demonstrates the need for adopting more realistic adversarial models when performing security analyses. Indeed, as we show later, the impact of forgeries generated under such conditions helps us to better understand the security of certain biometric-based schemes.

Our motivation for performing this analysis is primarily to show that there exists a disconnect between realistic threats and typical "best" practices [17] for reporting biometric performance—one that requires rethinking as both industry and the research community gains momentum in the exploration of biometric technologies. We believe that the type of analysis presented herein is of primary importance for the use of biometrics for authentication and cryptographic key generation (e.g., [21, 7, 2, 12]), where *weakest-link* analysis is paramount.

Moreover, to raise awareness of this shortcoming we explore a particular methodology in which we assume that the adversary utilizes indirect knowledge of the target user's biometric features. That is, we presume that the attacker has observed measurements of the biometric in contexts outside its use for security. For example, if the biometric is the user's handwriting dynamics generated while providing input via a stylus, then we presume the attacker has samples of the user's handwriting in another context, captured hardcopies of the user's writing, or writings from users of a similar style. We argue that doing so is more reflective of the real threats to biometric security. In this paper, we explore how an attacker can use such data to build *generative models* that predict how a user would, in this case, write a text, and evaluate the significance of this to biometric authentication.

2 Biometric Authentication

Despite the diversity of approaches examined by the biometrics community [1], from the standpoint of this investigation several key points remain relatively constant. For instance, the traditional procedure for applying a biometric as an authentication paradigm involves sampling an input from a user, extracting an appropriate set of features, and comparing these to previously stored templates to confirm or deny the claimed identity. While a wide range of features have been investigated, it is universally true that system designers seek features that exhibit large inter-class variability and small intra-class variability. In other words, two different users should be unlikely to generate the same input features, while a single user ought to be able to reproduce her own features accurately and repeatably.

Likewise, the evaluation of most biometric systems usually follows a standard model: enroll some number of users by collecting training samples, e.g., of their handwriting or speech. At a later time, test the rate at which users' attempts to recreate the biometric to within a predetermined tolerance fail. This failure rate is denoted as the False Reject Rate (`FRR`). Additionally, evaluation usually involves assessing the rate at which one user's input (i.e., an impostor) is able to fool the system when presented as coming from another user (i.e., the target). This evaluation yields the False Accept Rate (`FAR`) for the system under consideration. A tolerance setting to account for natural human variation is also vital in assessing the limits within which a sample will be consider as genuine, while at the same time, balancing the delicate trade-off of resistance to forgeries. Typically, one uses the equal error rate (`EER`)—that is, the point at which the `FRR` and the `FAR` are equal—to describe the accuracy of a given biometric system. Essentially, the lower the `EER`, the higher the accuracy.

Researchers also commonly distinguish between forgeries that were never intended to defeat the system ("random" or naïve forgeries), and those created by a user who was instructed to make such an attempt given information about the targeted input (i.e., so-called "skilled" forgeries). However, the evaluation of biometrics under such weak security assumptions can be misleading. Indeed, it may even be argued that because there is no strong means by which one can define a good forger and prove her existence (or non-existence), such analysis is theoretically impossible [29]. Nevertheless, the biometric community continues to rely on relatively simple measures of adversarial strength, and most studies to date only incorporate unskilled adversaries, and very rarely, "skilled" impersonators [13, 29, 11, 19, 20, 15].

This general practice is troubling as the evaluation of the `FAR` is likely to be significantly underestimated [29, 28]. Moreover, we believe that this relatively ad hoc approach to evaluation misses a significant threat: the use of *generative models* to create synthetic forgeries which can form the basis for sophisticated *automated* attacks on biometric security. This observation was recently reiterated in [32], where the authors conjectured that although the complexity of successful impersonations on various biometric modalities can be made formidable, biometric-based systems might be defeated using various strategies (see for example [31, 9, 26, 15]). As we show later, even rather simplistic attacks launched by successive replication of synthetic or actual samples from a representative population can have adverse effects on the `FAR`—particularly for the weakest users (i.e., the so-called "Lambs" in the biometric jargon for a hypothetical menagerie of users [3]).

In what follows, we provide what we believe is the most in-depth study to date that emphasizes the extent of this problem. Furthermore, as a first step towards providing system evaluators with a stronger methodology for quantifying performance under various threats, we describe our work on developing a prototype toolkit using handwriting dynamics as a case in point.

3 Handwriting Biometrics

Research on user authentication via handwriting has had a long, rich history, with hundreds of papers written on the topic. The majority of this work to date has focused on the problem of signature verification [27]. Signatures have some well known advantages: they are a natural and familiar way of confirming identity, have already achieved acceptance for legal purposes, and their capture is less invasive than most other biometric schemes [6]. While each individual has only one true signature—a notable limitation—handwriting in general contains numerous idiosyncrasies that might allow a writer to be identified.

In considering the mathematical features that can be extracted from the incoming signal to perform authentication, it is important to distinguish between two different classes of inputs. Data captured by sampling the position of a stylus tip over time on a digitizing tablet or pen computer are referred to as *online* handwriting, whereas inputs that are presented in the form of a 2-D bitmap (e.g., scanned off of a piece of paper) are referred to as *offline* handwriting. To avoid confusion with the traditional attack models in the security community, later on in this paper we shall eschew that terminology and refer to the former as covering both temporal and spatial information, whereas the latter only covers spatial information. Features extracted from offline handwriting samples include bounding boxes and aspect ratios, stroke densities in a particular region, curvature measurements, etc. In the online case, these features are also available and, in addition, timing and stroke order information that allows the computation of pen-tip velocities, accelerations, etc. Studies on signature verification and the related topic of handwriting recognition often make use of 50 or more features and, indeed, feature selection is itself a topic for research. The features we use in our own work are representative of those commonly reported in the field [8, 33, 18, 14]. Repeatability of features over time is, of course, a key issue, and it has been found that dynamic and static features are equally repeatable [8].

In the literature, performance figures (i.e., `EER`) typically range from 2% to 10% (or higher), but are difficult to compare directly as the sample sizes are often small and test conditions quite dissimilar [5]. Unfortunately, forgers are rarely employed in such studies and, when they are, there is usually no indication of their proficiency. Attempts to model attackers with a minimal degree of knowledge have involved showing a static image of the target signature and asking the impostor to try to recreate the dynamics [24]. The only serious attempt we are aware of, previous to our own, to provide a tool for training forgers to explore the limits of their abilities is the work by Zoebisch and Vielhauer [35]. In a small preliminary study involving four users, they found that showing an image of the target signature increased false accepts, and showing a dynamic replay doubled the susceptibility to forgeries yet again. However, since the verification algorithm used was simplistic and they do not report false reject rates, it is difficult to draw more general conclusions.

To overcome the "one-signature-per-user" (and hence, one key) restriction, we employ more general passphrases in our research. While signatures are likely to be more user-specific than arbitrary handwriting, results from the field of forensic analysis demonstrate that writer identification from a relatively small sample set is feasible [10]. Indeed, since this field focuses on handwriting extracted from scanned page images, the problem we face is less challenging in some sense since we have access to dynamic features in addition to static. Another concern, user habituation [5], is addressed by giving each test subject enough time to become comfortable with the experimental set-up and requiring practice writing before the real samples are collected. Still, this is an issue and the repeatability of non-signature passphrases is a topic for future research.

4 Experimental Design

We collected data over a two month period to analyze six different forgery styles. We consider three standard evaluation metrics: *naïve*, *static*, and *dynamic* [1] forgeries [13, 29, 11], as well as three metrics that will provide a more realistic definition of security: *naïve**, *trained*, and *generative*. Naïve, or "accidental", forgeries are not really forgeries in the traditional sense; they are measured by authenticating one user's natural writing samples of a passphrase against another user's template for the same passphrase. Static (resp. dynamic) forgeries are created by humans after seeing static (resp. real-time) renderings of a target user's passphrase. Naïve* forgeries are similar to naïve forgeries except that only writings from users of a similar style are authenticated against a target user's template. Trained forgeries are generated by humans under certain conditions, which will be described in greater detail later. Lastly, generative forgeries exploit information about a target user to algorithmically generate forgeries. Such information may include samples of the user's writing from a different context or general population statistics.

4.1 Data Collection

Our results are based on 11,038 handwriting samples collected on digitized pen tablet computers from 50 users during several rounds. We used NEC VersaLite Pad and HP Compaq TC1100 tablets as our writing platforms. The specifics of each round will be addressed shortly. To ensure that the participants were well motivated and provided writing samples reflective of their natural writing (as well as forgery attempts indicative of their innate abilities), several incentives were awarded for the most consistent writers, the best/most dedicated forgers, etc.

Data collection was spread across three rounds. In `round I`, we collected two distinct data sets. The first set established a baseline of "typical" user writing. After habituation on the writing device [5], users were asked to write five different phrases, consisting of two-word oxymorons, ten times each. We chose these phrases as they were easy to remember (and therefore, can be written naturally) and could be considered of reasonable length. Signatures were not used due to privacy concerns and strict restrictions on research involving human-subjects.

More importantly, in the context of key-generation, signatures are not a good choice for a hand-writing biometric as the compromise of keying material could prevent a user from using the system thereafter. This part of the data set was used for two purposes: to establish biometric templates to be used for authentication, and to provide samples for naive and naive* forgeries. To create a strong underlying representative system, users were given instructions to write as naturally (and consistently) as possible.

The second data set from `round I`, our "generative corpus", was used to create our generative forgeries and consisted of a set of 65 oxymorons. This set was restricted so that it did not contain any of the five phrases from the first data set, yet provided coverage of the first set at the bi-gram level. As before, we chose oxymorons that were easy to recall, and users were asked to write one instance of each phrase as naturally as possible. The average elapsed time for `round I` was approximately one hour.

`Round II` started approximately two weeks later. The same set of users wrote the five phrases from `round I` ten times. Additionally, the users were asked to forge representative samples (based on writing style, handedness of the original writer, and gender) from `round I` to create two sets of 17 forgeries. First, users were required to forge samples after seeing *only* a static representation. This data was used for our static forgeries. Next, users were asked to forge the same phrases again, but this time, upon seeing a real-time rendering of the phrase. At this stage, the users were instructed to make use of the real-time presentation to improve their rendering of the spatial features (for example, to distinguish between one continuous stroke versus two strokes that overlap) and to replicate the temporal features of the writing. This data comprised our dynamic forgeries. On average, `round II` took approximately 90 minutes for each user.

Lastly, in `round III` we selected nine users from `round II` who, when evaluated using the authentication system to be described in §4.2 and §4.3, exhibited a natural tendency to produce better forgeries than the average user in our study (although we did not include all of the best forgers). This group consisted of three "skilled" (but untrained) forgers for each writing style. (One of "cursive", "mixed", or "block", where the classification is based on the percent of the time that users connect adjacent characters.) Each skilled forger was asked to forge writing from the style which they exhibited an innate ability to replicate and was provided with a general overview and examples of the types of temporal and spatial characteristics that handwriting systems typically capture. As we were trying to examine (and develop) truly skilled adversaries, our forgers were asked to forge 15 writing samples from their specified writing style, with 60% of the samples coming from the weakest 10 targets, and the other 40% chosen at random. (In §5 we also provide the results of our trained forgeries against the entire population.) From this point on, these forgers (and their forgeries) will be referred to as "trained" forgers. We believe that the selection of the naturally skilled forgers, the additional training, and the selection of specific targets produced adversaries who realistically reflect a threat to biometric security.

The experimental setup for these educated forgers is as follows. First, a real-time reproduction of the target sample is displayed (at the top half of the tablet) and the forger is allowed to attempt forgeries (at her own pace) with the option of saving the attempts she liked. She can also select and replay her forgeries and compare them to the target. In this way, she is able to fine-tune her attempts by comparing the two writing samples. Next, she selects the forgery she believes to be her best attempt, and proceeds to the next target. The average elapsed time for this round was approximately two hours.

4.2 Authentication System

In order to have a concrete platform to measure the `FAR` for each of our six forgery styles, we loosely adapted the system presented in [34, 33] for generation of "biometric hashes". We note that our results are system-independent as we are only evaluating biometric *inputs*, for which we evaluated features that are reflective of the state of the art [14, 18, 8, 33].

For completeness, we briefly describe relevant aspects of the system; for a more detailed description see [33]. To input a sample to the system, a human writes a passphrase on an electronic tablet. The sample is represented as three signals parameterized by time. The discrete signals $x(t)$ and $y(t)$ specify the location of the pen on the writing surface at time t, and the binary signal $p(t)$ specifies whether the pen is up or down at time t. The tablet computes a set of n statistical features $(f_1, \ldots, f_n)$ over these signals. These features comprise the actual input to the authentication or key-generation system.

During an enrollment phase, each legitimate user writes a passphrase a pre-specified number (m) of times, and the feature values for each sample are saved. Let $f_{i,1}, \ldots, f_{i,n}$ denote the feature values for sample i. Using the feature values from each user and passphrase, the system computes a global set of tolerance values $(T = \{\epsilon_1, \ldots, \epsilon_n\})$ to be used to account for natural human variation [34]. Once the m readings have been captured, a biometric template is generated for each user and passphrase as follows: Let $\ell'_j = \min_{i\in[1,m]} f_{i,j}$, $h'_j = \max_{i\in[1,m]} f_{i,j}$, and $\Delta_j = h'_j - \ell'_j + 1$. Set $\ell_j = \ell'_j - \Delta_j\epsilon_j$, and $h_j = h'_j + \Delta_j\epsilon_j$. The resulting template is an $n \times 2$ matrix of values $\{\{\ell_1, h_1\}, \ldots, \{\ell_n, h_n\}\}$.

Later, when a user provides a sample with feature values $f_1, \ldots, f_n$ for authentication, the system checks whether $f_j \in [\ell_j, h_j]$ for each feature f_j. Each $f_j \notin [\ell_j, h_j]$ is deemed an error, and depending on the threshold of errors tolerated by the system, the attempt is either accepted or denied. We note that as defined here, templates are insecure because they leak information about a user's feature values. We omit discussion of securely representing biometric templates (see for example [22, 4]) as this is not a primary concern of this research.

4.3 Feature Analysis

Clearly, the security of any biometric system is directly related to the quality of the underlying features. A detailed analysis of proposed features for handwriting verification is presented in [33], although we argue that the security model of that work sufficiently differs from our own and so we believe a new feature-evaluation metric was required. In that work, the quality of a feature was measured by the deviation of the feature and entropy of the feature across the population. For our purposes, these evaluation metrics are not ideal: we are not only concerned with the entropy of each feature, but rather how difficult the feature is to *forge* — which we argue is a more important criteria. When systems are evaluated using purely naïve forgeries, then entropy could be an acceptable metric. However, as we show later, evaluation under naïve forgeries is not appropriate [2].

As our main goal is to highlight limitations in current practices, we needed to evaluate a robust yet usable system based on a strong feature set. To this end, we implemented 144 state of the art features [33, 8, 25, 14] and evaluated each based on a quality metric (Q) defined as follows. For each feature f, we compute the proportion of times that f was missed by legitimate users in our study, denoted r_f, and the proportion of times that f was missed by forgers from `round II` (with access to dynamic information), denoted a_f. Then, $Q(f) = (a_f - r_f + 1)/2$, and the range of Q is $[0, 1]$. Intuitively, features with a quality score of 0 are completely useless—i.e., they are *never* reliably reproduced by original users and are *always* reproduced by forgers. On the other hand, features with scores closer to 1 are highly desirable when implementing biometric authentication systems.

For our evaluation, we divided our feature set into two groups covering the temporal and spatial features, and ordered each according to the quality score. We then chose the top 40 from each group, and disregarded any with a `FRR` greater than 10%. Finally, we discounted any features that could be inferred from others (e.g., given the width and height of a passphrase as rendered by a user, then a feature representing the ratio between width and height is redundant). This analysis resulted in what we deem the 36 best features—15 spatial and 21 temporal—described in Appendix A.

5 Human Evaluation

This section presents the results for the five evaluation metrics that use forgeries generated by humans. Before we computed the `FRR` and the `FAR`, we removed the outliers that are inherent to biometric systems. For each user, we removed all samples that had more than $\delta = 3$ features that fell outside $k = 2$ standard deviations from that user's mean feature value. The parameters δ and k were empirically derived. We also did not include any samples from users (the so-called "Goats" [3]) who had more than 25% of their samples classified as outliers. Such users "Failed to Enroll" [17]; the FTE rate was $\approx 8.7\%$. After combining this with outlier removal, we still had access to 79.2% of the original data set.

To compute the `FRR` and the `FAR` we use the system described in §4.2 using the features from §4.3. The `FRR` is computed as follows: we repeatedly randomly partition a user's samples into two groups and use the first group to build a template and authenticate the samples in the second group against the template. To compute the `FAR` we use all of the user's samples to generate a template and then authenticate the forgeries against this template.

5.1 Grooming Sheep into Wolves

Our experiments were designed to illustrate the discrepancy in perceived security when considering traditional forgery paradigms and a more stringent, but realistic, security model. In particular, we assume that at the very minimum, that a realistic adversary (1) attacks victims who have a writing style that the forger has a natural ability to replicate, (2) has knowledge of how biometric authentication systems operate, and (3) has a vested interest in accessing the system, and therefore is willing to devote significant effort towards these ends.

Figure 1 presents ROC curves for forgeries from impersonators with varying levels of knowledge. The plot denoted `FAR`-naïve depicts results for the traditional case of naïve forgeries widely used in the literature [13, 29, 11]. In these cases, the impersonation attempts simply reflect taking one user's natural rendering of phrase p as an impersonation attempt on the target writing p. Therefore, in addition to ignoring the target writer's attributes as is naturally expected of forgers, this classification makes no differentiation based on the forger's or the victim's style of writing, and so may include, for example, block writers "forging" cursive writers. Arguably, such forgeries may not do as well as the less standard (but more reasonable) naïve* classification (FAR-naïve*) where one only attempts to authenticate samples from writers of similar styles.

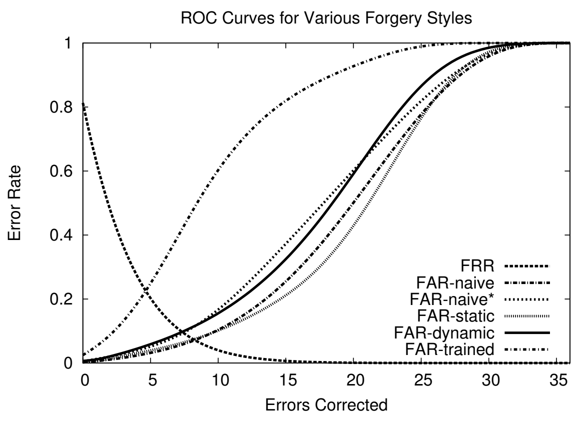


Figure 1: Overall ROC curves for naïve, naïve*, static, dynamic, and trained forgers.

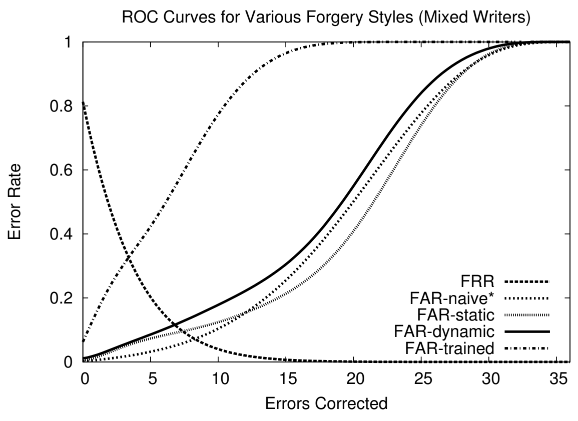


Figure 2: ROC curves against all *mixed* writers. This grouping appeared the easiest to forge by the users in our study.

The FAR-static plots represent the success rate of forgers who receive access to only a static rendering of the passphrase. By contrast, FAR-dynamic forgeries are produced after seeing (possibly many) real-time renderings of the image. One can easily consider this a realistic threat if we assume that a motivated adversary may capture the writing on camera, or more likely, may have access to data written electronically in another context. Lastly, FAR-trained presents the resulting success rate of forgeries derived under our forgery model which captures a more worthy opponent. Notice that when classified by writing style, the trained forgers were very successful against mixed writers (Figure 2).

Intuitively, one would expect that forgers with access to dynamic and/or static representations of the target writing should be able to produce better forgeries than those produced under the naïve* classification. This is not necessarily the case, as we see in Figure 1 that at some points, the naïve* forgeries do better than the forgeries generated by forgers who have access to static and/or dynamic information. This is primarily due to the fact that the naïve* classification reflects users' normal writing (as there is really no forgery attempt here). The natural tendencies exhibited in their writings appear to produce better "forgeries" than that of static or dynamic forgers (beyond some point), who may suffer from unnatural writing characteristics as a result of focusing on the act of forging.

One of the most striking results depicted in the figures is the significant discrepancy in the FAR between standard evaluation methodologies and that of the trained forgeries captured under our strengthened model. While it is tempting to directly compare the results under the new model to those under the more traditional metrics (i.e., by contrasting the FAR-trained error rate at the EER under one of the older models), such a comparison is *not* valid. This is because the forgers under the new model were more knowledgeable with respect to the intricacies of handwriting verification and had performed style-targeted forgeries.

However, the correct comparison considers the EERs under the two models. For instance, the EER for this system under FAR-trained forgeries is approximately 20.6% at four error corrections. However, for the more traditional metrics, one would arrive at EERs of 7.9%, 6.0%, 5.5% under evaluations of dynamic, static and naïve forgeries, respectively. These results are indeed inline with the current state of the art [13, 29, 11]. Even worse, under the most widely used form of adversary considered in the literature (i.e., naïve) we see that the security of this system would be over-estimated by nearly 375%!

Forger Improvement Figure 3 should provide assurance that the increase in forgery quality is not simply a function of selecting naturally skilled individuals from round II to participate in round III. The graph shows the improvement in FAR between rounds II and III for the trained forgers. We see that the improvement is significant, especially for the forgers who focused on mixed and block writers. Notice that at the EER (at seven errors) induced by forgers with access to dynamic information (Figure 1), our trained mixed, block, and cursive forgers improved their FAR by 0.47, 0.34, and 0.18, re-

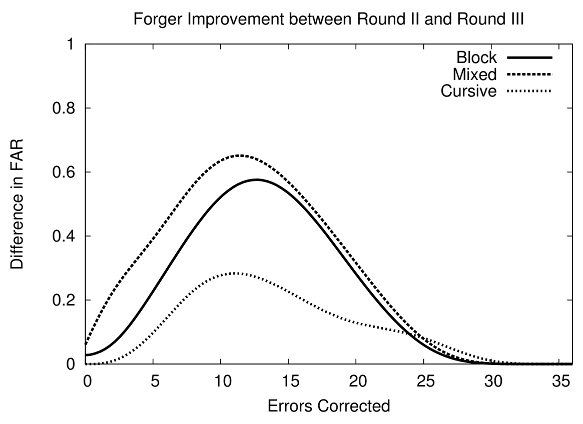


Figure 3: Forger improvement between `rounds II` and `III`.

spectively. This improvement results from less than two hours of training and effort, which is likely much less than what would be exerted by a dedicated or truly skilled forger.

The observant reader will note that the trained forgers faced a different distribution of "easy" targets in Round `III` then they did in Round `II`. We did this to analyze the system at its weakest link. However, after normalizing the results so that both rounds had the same makeup of "easy" targets, the change in `EER` is statistically insignificant, shifting from 20.6% to 20.0% at four errors corrected.

6 Generative Evaluation

Unfortunately, finding and training "skilled" forgers is a time (and resource) consuming endeavor. To confront the obstacles posed by wide-scale data collection and training of good impersonators, we decided to explore the use of an automated approach using generative models as a supplementary technique for evaluating behavioral biometrics. We investigated whether an automated approach, using limited writing samples from the target, could match the false accept rates observed for our trained forgers in §5.1. We believe that such generative attacks themselves may be a far more dangerous threat that, until now, have yet to be studied in sufficient detail.

For the remaining discussion we explore a set of threats that stem from generative attacks which assume knowledge that spans the following spectrum:

I. *General population statistics*: Gleaned, for example, via the open sharing of test data sets by the research community, or by recruiting colleagues to provide writing samples.

II. *Statistics specific to a demographic of the targeted user*: In the case of handwriting, we assume the attacker can extract statistics from a corpus collected from other users of a similar writing style (e.g., cursive).

III. *Data gathered from the targeted user*: Excluding direct capture of the secret itself, one can imagine the attacker capturing copies of a user's handwriting, either through discarded documents or by stealing a PDA.

To make this approach feasible, we also explore the impact of these varying threats. A key issue that we consider is the amount of recordings one needs to make these scenarios viable attack vectors. As we show later, the amount of data required may be surprisingly small for the case of authentication systems based on handwriting dynamics.

6.1 A generative toolkit for performance testing

The approach to synthesizing handwriting we explore here is to assemble a collection of basic units (n-grams) that can be combined in a concatenative fashion to mimic authentic handwriting. In this case, we do not make use of an underlying model of human physiology, rather, creation of the writing sample is accomplished by choosing appropriate n-grams from an inventory that may cover writing from the target user (scenario III above) as well as representative writings by other members of the population at large (scenarios I and II). The technique we apply here expands upon earlier rudimentary work [16], and is similar in flavor to approaches taken to generate synthesized speech [21] and for text-to-handwriting conversion [9].

6.1.1 Forgeries

As noted earlier, each writing sample consists of three signals parameterized by time: $x(t)$, $y(t)$ and $p(t)$. The goal of our generative algorithm is to generate t, $x(t)$, $y(t)$ and $p(t)$ such that the sample is not only accepted as authentic, but relies on acquiring a minimal amount of information from the target user (again, in a different security context). In particular, when attacking user u, we assume the adversary has access to a generative corpus $\mathcal{G}_u$, in addition to samples from users of similar writing styles $\mathcal{G}_S$; where S is one of "block", "mixed", or "cursive". We assume that both $\mathcal{G}_u$ and $\mathcal{G}_S$ are annotated so that there is a bijective map between the characters of each phrase and the portion of the signal that represents each character. As is the case with traditional compu-

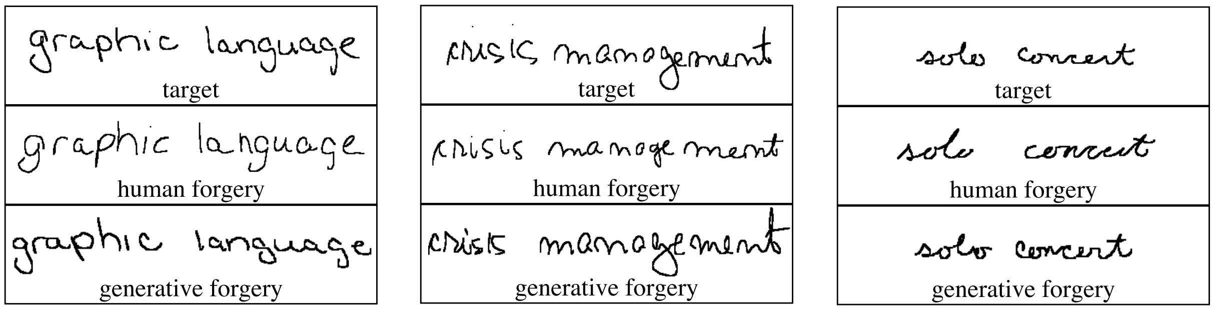


Figure 4: Example generative forgeries against block, mixed and cursive forgers. For each box, the second rendering is a trained human-generated forgery of the first, and the third was created by our generative algorithm.

tations of the EER we also assume that passphrase p is known.

General Knowledge Assume that the adversary wishes to forge user u with passphrase p and writing style S. Ideally, she would like to do so using a minimal amount of information directly collected from u. Fortunately, the success of the naïve* forgeries from §5 suggests that a user's writing style yields a fair amount of pertinent information that can potentially be used to replicate that user's writing. Thus, to aid in generating accurate forgeries, the adversary can make use of several statistics computed from annotated writing samples in $\mathcal{G}_S \backslash \mathcal{G}_u$. In what follows, we discuss what turn out to be some very useful measures that can likely be easily generalized for other behavioral biometrics.

Denote as $P_c(i, j, c_1, c_2)$ the probability that writers of style S connect the i^{th} stroke of c_1 to c_2, given that c_1 is comprised of j strokes. Let $P_c(i, j, c_1, *)$ be the probability that these writers connect the i^{th} stroke of c_1 (again rendered with j strokes) to any adjacent letter. For example, many cursive writers will connect the first stroke of the letter 'i' to proceeding letters; for such writers $P_c(1, 2, \texttt{i}, *) \approx 1$. Note that in this case, the dot of the 'i' will be rendered after proceeding letters, we call this a "delayed" stroke.

Let $\delta_w(c_1, c_2)$ denote the median gap between the adjacent characters c_1 and c_2 (i.e., the distance between the maximum value of $x(t)$ for c_1 and the minimum value of $x(t)$ for c_2), $\delta_w(c_1, *)$ the median gap between c_1 and any proceeding character, and $\delta_w(*)$ the median gap between any two adjacent characters. Intuitively, $\delta_w(c_1, c_2) < 0$ if users tend to overlap characters. Similarly, let $\delta_t(c_1, c_2)$ denote the median time elapsed between the end of c_1 and the beginning of c_2. Definitions of $\delta_t(c_1, *)$ and $\delta_t(*)$ are analogous to those for δ_w.

Finally, the generative algorithm clearly must also make use of a user's pen-up velocity. This can be estimated from the population by computing the pen-up velocity for each element in $\mathcal{G}_S$ and using the 75th percentile of these velocities. We denote this value as v_S.

Having acquired her generalized knowledge, the adversary can now select and combine her choices of n-grams that will be used for concatenative-synthesis in the following manner:

n-gram Selection At a high level, the selection of n-grams that allow for a concatenative-style rendering of p involves a search of $\mathcal{G}_u$ for possible candidates. Let $\mathcal{G}_{u,p}$ be a set of u's renderings of various n-grams in p. There may be more than one element in $\mathcal{G}_{u,p}$ for each n-gram in p. The attacker selects k renderings $g_1, \ldots, g_k$ from $\mathcal{G}_{u,p}$ such that $g_1||g_2||\ldots||g_k = p$. Our selection algorithm is randomized, but biased towards longer n-grams. However, the average length of each n-gram is small as shorter n-grams are required to "fill the gap" between larger n-grams. To explore the feasibility of our generative algorithm we ensure that g_i and g_{i+1} do not originate from the same writing sample, but an actual adversary might benefit from using n-grams from the same writing sample.

n-gram Combination Given the selection of n-grams $(g_1, \ldots, g_k)$ the attacker's task is to combine them to form a good representation of p. Namely, she must adjust the signals that compose each g_i (t_{g_i}, $x(t_{g_i})$, $y(t_{g_i})$ and $p(t_{g_i})$) to create a final set of signals that authenticates to the system. The algorithm is quite simple. At a high level, it proceeds as follows: The adversary normalizes the signals t_{g_i}, $x(t_{g_i})$ and $y(t_{g_i})$ by subtracting the respective minimum values from each element in the signal. The $y(t_{g_i})$ are shifted so that the baselines of the writing match across g_i. To finalize the spatial transforms, the adversary horizontally shifts each $x(t_{g_i})$ by

$$\delta_{x,i} = \delta_{x,i-1} + \max(x(t_{g_{i-1}})) + \delta_w(e_{i-1}, s_i)$$

where e_i (resp. s_i) is the last (resp. first) character in g_i and $\delta_{x,1} = 0$. Once the adversary has fixed the (x, y) coordinates, she needs to fabricate t and $p(t)$ signals to complete the forgery. Modifying $p(t)$ consists of deciding whether or not to connect adjacent n-grams. To do this, the adversary uses knowledge derived from the population. If e_{i-1} is rendered with j' strokes, and g_i starts with s_i, the adversary connects the j^{th} stroke of e_{i-1} to s_i with probability $P_c(j, j', e_{i-1}, s_i)$. To generate a more realistic connection, the adversary smoothes the last points of e_{i-1} and the first points of s_i. Additionally, all strokes that occur after stroke j are "pushed" onto a stack, which is emptied on the next generated pen-up. This behavior simulates a true cursive writer returning to dot 'i's and cross 't's at the end of a word, processing characters closest to the end of the word first.

Adjusting the t signal is also straightforward. Let T be the time in $t_{g_{i-1}}$ that the last non-delayed stroke in e_{i-1} ends. If there are no delayed strokes in e_{i-1}, $T = \max(t_{g_{i-1}})$. Then, the adversary can simply shift t_{g_i}, $i > 1$ by

$$\delta_{\tau,i} = \delta_{\tau,i-1} + T + \delta_t(e_{i-1}, s_i)$$

and $\delta_{\tau,1}$ = 0. The only other time shift occurs when delayed strokes are popped from the stack. We can make use of global knowledge to estimate the time delay by using v_S and the distance between the end of the previous stroke and the new stroke. Note that it is beneficial to take v_S as the 75th percentile instead of the median velocity because, for cursive writers in particular, the majority of pen-up velocities is dominated by the time between words. However, these velocities are intuitively slower as the writer is now thinking about creating a new word as opposed to finishing a word that already exists.

If the adversary does not have access to the statistical measure $\delta_w(e_{i-1}, s_i)$, she can first base her estimate of inter-character spacing on $\delta_w(e_{i-1}, *)$, and then on $\delta_w(*)$. She proceeds similarly for the measures δ_t and P_c.

6.2 Results

To evaluate this concatenative approach we analyzed the quality of the generated forgeries on user u writing passphrase p. However, rather than using all 65 of the available samples from the generative corpus, we instead choose 15 samples at random from $\mathcal{G}_{u,p}$ — with the one restriction being that there must exist at least one instance of each character in p among the 15 samples. Recall that this generative corpus contains writing samples from u, but does not include p. The attacker's choice of n-grams $g_1, \ldots, g_k$ are selected from this restricted set.

Additionally, we limit $\mathcal{G}_S$ to contain only 15 randomly selected samples from each user with a similar writing style as u. Denote this set of writings as $\mathcal{G}'_S$. We purposefully chose to use small (and arguably, easily obtainable) data sets to illustrate the power of this concatenative attack. Our "general knowledge" statistics are computed from $\mathcal{G}'_S$. Example forgeries derived by this process are shown in Figure 4.

We generated up to 25 forgery attempts for each user u and phrase p and used each as an attempt to authenticate to the biometric template corresponding to u under p. Figure 5 depicts the average `FAR` across all forgery attempts. As a baseline for comparison, we replot the `FRR` and the `FAR`-trained plots from §5. The `FAR`-generative plot shows the results of the generative algorithm against the entire population. Observe that under these forgeries there is an `EER` of 27.4% at three error correction compared to an `EER` of 20.6% at four error corrections when considering our trained forgers.

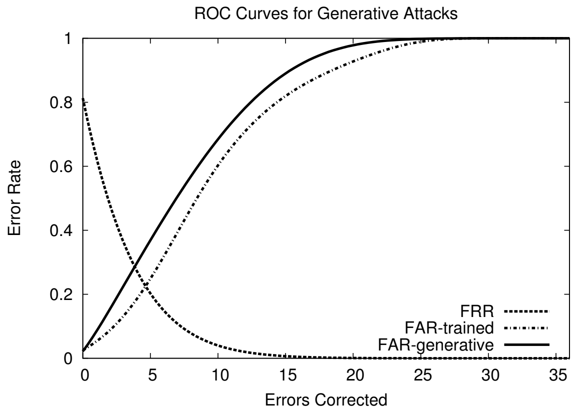


Figure 5: ROC curves for generative forgeries. Even with access to only limited information, the algorithm out-performs our trained forgers, shifting the `EER` from 20.6% at four errors to 27.4% at three errors.

We note that on average each generative attempt only used information from 6.67 of the target user's writing samples. Moreover, the average length of an n-gram was 1.64 characters (and was never greater than 4). More importantly, as we make no attempt to filter the output of the generative algorithm by rank-ordering the best forgeries, the results could be much improved. That said, we believe that given the limited information assumed here, the results of this generative attack on the security of the system warrant serious consideration. Furthermore, we believe that this attack is feasible because annotation of the samples in $\mathcal{G}_{u,p}$, while tedious, poses only a minor barrier to any determined adversary. For instance, in our case annotation was accomplished with the aide of an annotation tool that we implemented which is fairly automated, especially for block handwriting: taking $\approx$ 30 sec. to annotate block phrases and $\approx$ 1.5 min. for cursive phrases.

7 Other Related Work

There is, of course, a vast body of past work on the topic of signature verification (see [27] for a comprehensive if somewhat dated survey, [11] for a more up-to-date look at the field). However, to the best of our knowledge, there is relatively little work that encompass our goals and attack models described herein.

Perhaps the work closest to ours, although it predominately involves signatures, is that by Vielhauer and Steinmetz [33]. They use 50 features extracted from a handwriting sample to construct a biometric hash. While they performed some preliminary testing on PIN's and passphrases, the bulk of their study is on signatures, where they evaluated features based on intrapersonal deviation, interpersonal entropy with respect to their hash function, and the correlation between these two values. That work however does not report any results for meaningful attempts at forgery (i.e, other than naïve attacks).

Also germane are a series of recent papers that have started to examine the use of dynamic handwriting for the generation of cryptographic keys. Kuan, et al. present a method based on block-cipher principles to yield cryptographic keys from signatures [12]. They test their algorithm on the standard data set from the *First International Signature Verification Competition* and report `EER`s between 6% and 14% if the forger has access to a stolen token. The production of skilled forgeries in the SVC data set [37] resembles part of the methodology used in `round II` of our studies and so does not account for motivation, training, or talent.

In the realm of signature verification we also note work on an attack based on hill-climbing, but that makes the assumption that the system reveals how close of a match the input is [36]. We believe this to be clearly unrealistic, and our attack models are chosen to be more pragmatic than this.

Finally, there have been a handful of works on using generative models to attack biometric authentication. However, we note there exists significant disagreement in the literature concerning the potential effectiveness of similar (but inherently simpler) attacks on speaker verification systems (e.g., [26, 21]). Lindberg and Blomberg, for example, determined that synthesized passphrases were not effective in their small-scale experiments [15], whereas Masuko et al. found that their system was easily defeated [20].

8 Conclusions

Several fundamental computer security mechanisms rest on the ability of an intended user to generate an input that an attacker is unable to reproduce. In the biometric community, the security of biometric-based technologies hinges on this perceived inability of the attacker to reproduce the target user's input. In particular, the evaluation of biometric technologies is usually conducted under fairly weak adversarial conditions. Unfortunately, this practice may significantly underestimate the real risk of accepting forgeries as authentic. To directly address this limitation we present an automated technique for producing generative forgeries that assists in the evaluation of biometric systems. We show that our generative approach matches or exceeds the effectiveness of forgeries rendered by trained humans in our study.

Our hope is that this work will serve as a solid foundation for the work of other researchers and practitioners, particularly as it pertains to evaluating biometric authentication or key-generation systems. Admittedly, such evaluations are difficult to undertake due to the reliance of recruiting large numbers of human subjects. In that regard, the generative approach presented herein should reduce the difficulty of this task and allow for more rigorous evaluations as it pertains to biometric security.

Additionally, there is much future work related to the topics presented here. For instance, although the forgeries generated by our trained forgers were alarmingly successful, it remains unclear as to the extent to which these forgeries would fool human judges, including for example, forensic document examiners. Exploring this question is one of our short term goals. Lastly, there are several directions for incorporating more sophisticated generative algorithms into our evaluation paradigm. We hope to explore these in the coming months.

Acknowledgments

The authors would like to thank Dishant Patel and Carolyn Buckley for their help in our data collection efforts. We especially thank the many people who devoted hours to providing us with handwriting samples. We thank the anonymous reviewers, and in particular, our shepherd Tara Whalen, who provided helpful suggestions for improving this paper. We also thank Michael K. Reiter for insightful discussions during the course of this research. This work is supported by NSF grant CNS-0430338.

Notes

[1]Although the biometric literature often refers to static or dynamic forgeries as skilled forgeries, here we make a distinction between these three types. For example, despite access to static or dynamic information, a weak forger might not be able to successfully replicate another user's writing.

[2]It is interesting to note, however, that each strong feature as defined in [33] may be inferred from our best features. However, we did find several other features that were not included in the original work.

References

[1] The biometrics consortium. http://www.biometrics.org/.

[2] Y.-J. Chang, W. Zhung, and T. Chen. Biometrics-based cryptographic key generation. In *Proceedings of the International Conference on Multimedia and Expo*, volume 3, pages 2203–2206, 2004.

[3] G. R. Doddington, W. Liggett, A. F. Martin, M. Przybocki, and D. A. Reynolds. Sheep, goats, lambs and wolves: A statistical analysis of speaker performance in the NIST 1998 speaker recognition evaluation. In *Proceedings of the Fifth International Conference on Spoken Language Processing*, November 1998.

[4] Y. Dodis, L. Reyzin, and A. Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In *Advances in Cryptology—EUROCRYPT 2004*, pages 523–540, 2004.

[5] S. J. Elliott. Development of a biometric testing protocol for dynamic signature verification. In *Proceedings of the International Conference on Automation, Robotics, and Computer Vision*, pages 782–787, Singapore, 2002.

[6] M. C. Fairhurst. Signature verification revisited: promoting practical exploitation of biometric technology. *Electronics & Communication Engineering Journal*, pages 273–280, December 1997.

[7] A. Goh and D. C. L. Ngo. Computation of cryptographic keys from face biometrics. In *Proceedings of Communications and Multimedia Security*, pages 1–13, 2003.

[8] R. M. Guest. The repeatability of signatures. In *Proceedings of the Ninth International Workshop on Frontiers in Handwriting Recognition*, pages 492–497, October 2004.

[9] I. Guyon. Handwriting synthesis from handwritten glyphs. In *Proceedings of the Fifth International Workshop on Frontiers of Handwriting Recognition*, pages 140–153, Colchester, England, 1996.

[10] C. Hertel and H. Bunke. A set of novel features for writer identification. In *Proceedings of the International Conference on Audio- and Video-based Biometric Person Authentication*, pages 679–687. Guilford, UK, 2003.

[11] A. K. Jain, F. D. Griess, and S. D. Connell. On-line signature verification. *Pattern Recognition*, 35(12):2963–2972, 2002.

[12] Y. W. Kuan, A. Goh, D. Ngo, and A. Teoh. Cryptographic keys from dynamic hand-signatures with biometric security preservation and replaceability. In *Proceedings of the Fourth IEEE Workshop on Automatic Identification Advanced Technologies*, pages 27–32, Los Alamitos, CA, 2005. IEEE Computer Society.

[13] F. Leclerc and R. Plamondon. Automatic signature verification: the state of the art 1989-1993. *International Journal of Pattern Recognition and Artificial Intelligence*, 8(3):643–660, 1994.

[14] L. Lee, T. Berger, and E. Aviczer. Reliable on-line human signature verification systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(6):643–647, June 1996.

[15] J. Lindberg and M. Blomberg. Vulnerability in speaker verification – a study of technical impostor techniques. In *Proceedings of the European Conference on Speech Communication and Technology*, volume 3, pages 1211–1214, Budapest, Hungary, September 1999.

[16] D. P. Lopresti and J. D. Raim. The effectiveness of generative attacks on an online handwriting biometric. In *Proceedings of the International Conference on Audio- and Video-based Biometric Person Authentication*, pages 1090–1099. Hilton Rye Town, NY, USA, 2005.

[17] A. J. Mansfield and J. L. Wayman. Best practices in testing and reporting performance of biometric devices. Technical Report NPL Report CMSC 14/02, Centre for Mathematics and Scientific Computing, National Physical Laboratory, August 2002.

[18] U.-V. Marti, R. Messerli, and H. Bunke. Writer identification using text line based features. In *Proceedings of the Sixth International Conference on Document Analysis and Recognition*, pages 101–105, September 2001.

[19] T. Masuko, T. Hitotsumatsu, K. Tokuda, and T. Kobayashi. On the security of hmm-based speaker verification systems against imposture using synthetic speech. In *Proceedings of the European Conference on Speech Communication and Technology*, volume 3, pages 1223–1226, Budapest, Hungary, September 1999.

[20] T. Masuko, K. Tokuda, and T. Kobayashi. Imposture using synthetic speech against speaker verification based on spectrum and pitch. In *Proceedings of the International Conference on Spoken Language Processing*, volume 3, pages 302–305, Beijing, China, October 2000.

[21] F. Monrose, M. Reiter, Q. Li, D. Lopresti, and C. Shih. Towards speech-generated cryptographic keys on resource-constrained devices. In *Proceedings of the Eleventh USENIX Security Symposium*, pages 283–296, 2002.

[22] F. Monrose, M. K. Reiter, Q. Li, and S. Wetzel. Cryptographic key generation from voice (extended abstract). In *Proceeedings of the 2001 IEEE Symposium on Security and Privacy*, pages 12–25, May 2001.

[23] F. Monrose, M. K. Reiter, and S. Wetzel. Password hardening based on keystroke dynamics. *International Journal of Information Security*, 1(2):69–83, February 2002.

[24] I. Nakanishi, H. Sakamoto, Y. Itoh, and Y. Fukui. Optimal user weighting fusion in DWT domain on-line signature verification. In *Proceedings of the International Conference on Audio- and Video-based Biometric Person Authentication*, pages 758–766. Hilton Rye Town, NY, USA, 2005.

[25] W. Nelson and E. Kishon. Use of dynamic features for signature verification. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, pages 1504–1510, October 1991.

[26] B. L. Pellom and J. H. L. Hansen. An experimental study of speaker verification sensitivity to computer voice altered imposters. In *Proceedings of the 1999 International Conference on Acoustics, Speech, and Signal Processing*, March 1999.

[27] R. Plamondon, editor. *Progress in Automatic Signature Verification*. World Scientific, 1994.

[28] R. Plamondon and G. Lorette. Automatic signature verification and writer identification – the state of the art. volume 22, pages 107–131, 1989.

[29] R. Plamondon and S. N. Srihari. On-line and off-line handwriting recognition: a comprehensive survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1):63–84, 2000.

[30] C. Soutar, D. Roberge, A. Stoianov, R. Gilroy, and B. V. Kumar. Biometric encryption^{TM} using image processing. In *Optical Security and Counterfeit Deterrence Techniques II*, volume 3314, pages 178–188. IS&T/SPIE, 1998.

[31] U. Uludag and A. K. Jain. Fingerprint minutiae attack system. In *The Biometric Consortium Conference*, September 2004.

[32] U. Uludag, S. Pankanti, S. Prabhakar, and A. K. Jain. Biometric cryptosystems: Issues and challenges. *Proceedings of the IEEE: Special Issue on Multimedia Security of Digital Rights Management*, 92(6):948–960, 2004.

[33] C. Vielhauer and R. Steinmetz. Handwriting: Feature correlation analysis for biometric hashes. *EURASIP Journal on Applied Signal Processing*, 4:542–558, 2004.

[34] C. Vielhauer, R. Steinmetz, and A. Mayerhofer. Biometric hash based on statistical features of online signatures. In *Proceedings of the Sixteenth International Conference on Pattern Recognition*, volume 1, pages 123–126, 2002.

[35] C. Vielhauer and F. Zöbisch. A test tool to support brute-force online and offline signature forgery tests on mobile devices. In *Proceedings of the International Conference on Multimedia and Expo*, volume 3, pages 225–228, 2003.

[36] Y. Yamazaki, A. Nakashima, K. Tasaka, and N. Komatsu. A study on vulnerability in on-line writer verification system. In *Proceedings of the Eighth International Conference on Document Analysis and Recognition*, pages 640–644, Seoul, South Korea, August-September 2005.

[37] D.-Y. Yeung, H. Chang, Y. Xiong, S. George, R. Kashi, T. Matsumoto, and G. Rigoll. SVC2004: First international signature verification competition. In *Proceedings of the International Conference on Biometric Authentication (ICBA)*, Hong Kong, July 2004.

A Features

Using the quality metric, Q, as described in §4.3 we narrowed 144 state of the art features to the 36 most useful features (see Table 1). The 15 static features consisted of: the number of strokes used in rendering the phrase, the number of local horizontal and vertical extrema, and the integrated area to the left and below the writing [33]. Additional static features included the writing width and height, the total distance travelled by the pen on and off the tablet, the total area enclosed within writing loops, and the vertical centroid of these loops [8]. We also considered the distance between the upper (lower) baseline and the top (bottom) line [18], the median stroke-slant [18], and the distance between the last x (y) coordinate and the maximum x (y) coordinate [14]. Note that these final two features could be considered dynamic as one may not know which coordinate is the last one rendered without access to timing information.

The 21 dynamic features consisted of: The total time spent writing, the ratio of pen-up time to pen-down time, the median pen velocity, the number of times the pen ceases to move horizontally (vertically), and the total time spent moving to the left, right, up, and down [14]. Additional dynamic features included the time of occurrence of the following events: maximum pen velocity, maximum pen velocity in the horizontal (vertical) direction, minimum velocity in the horizontal (vertical) direction, and the maximum stroke slant [14]. Finally, we considered six invariant moments of the writing, which measure the number of samples, horizontal (vertical) mass, diagonality, and horizontal (vertical) divergence [8].

Feature (f)	Description	$Q(f)$
	Spatial Features	
Pen-down distance	Total distance travelled by the pen-tip while touching the screen [8].	0.81
Median θ	Median stroke-slant, normalized to $\theta \in [0, \pi]$ [18].	0.71
Vert. end dist.	Distance between the last y-coordinate and maximum y-coordinate [14].	0.67
Y-Area	Integrated area beneath the writing [33].	0.65
Writing width	Total width of the writing [33, 8].	0.65
Writing height	Total height of the writing [33, 8].	0.65
Pen-up distance	Euclidean distance between pen-up and pen-down events.	0.64
# of strokes	Number of strokes used to render the passphrase [33].	0.63
# of extrema	Number of local extrema in the horizontal and vertical directions [33].	0.62
Lower zone	Distance between baseline and bottomline of the writing [18].	0.62
X-Area	Integrated area to the left of the writing [33].	0.62
Loop y centroid	The average value of all y coordinates contained within writing loops [8].	0.62
Loop area	Total area enclosed within loops generated by overlapping strokes [8].	0.61
Upper zone	Distance between upper-baseline and topline of the writing [18].	0.61
Horiz. end dist.	Distance between the last x-coordinate and maximum x-coordinate [14].	0.60
	Temporal Features	
Time	Total time spent writing (measured in ms) [14].	0.87
# of times $v_x = 0$	Number of times the pen ceases to move horizontally [14].	0.86
# of times $v_y = 0$	Number of times the pen ceases to move vertically [14].	0.85
Inv. Mom. 00	$\sum_x \sum_y f(x, y)$; $f(x, y) = 1$ if there is a point at (x, y) and 0 otherwise [8].	0.85
Inv. Mom. 10	$\sum_x \sum_y f(x, y) \cdot x$. Measures the horizontal mass of the writing [8].	0.82
Inv. Mom. 01	$\sum_x \sum_y f(x, y) \cdot y$. Measures the vertical mass of the writing [8].	0.79
Inv. Mom. 11	$\sum_x \sum_y f(x, y) \cdot xy$. Measures diagonality of the writing sample [8].	0.78
Time of max v_x	Time of the maximum pen-velocity in the horizontal direction [14].	0.78
Inv. Mom. 21	$\sum_x \sum_y f(x, y) \cdot x^2 y$. Measures vertical divergence [8].	0.76
Inv. Mom. 12	$\sum_x \sum_y f(x, y) \cdot xy^2$. Measures horizontal divergence [8].	0.75
Median pen velocity	Median speed of the pen-tip [14].	0.74
Duration $v_x > 0$	Total time the pen spends moving to the right [14].	0.73
Duration $v_y > 0$	Total time the pen spends moving to the up [14].	0.73
Time of max vel.	Time of the maximum pen-velocity [14].	0.72
Pen up/down ratio	Ratio time spent with the pen off and on the tablet [14].	0.71
Time of max θ	Time of maximum stroke slant.	0.70
Duration $v_y < 0$	Total time the pen spends moving to the down [14].	0.70
Duration $v_x < 0$	Total time the pen spends moving to the left [14].	0.69
Time of min v_x	Time of the minimum pen-velocity in the horizontal direction [14].	0.69
Time of min v_y	Time of the minimum pen-velocity in the vertical direction [14].	0.68
Time of max v_y	Time of the maximum pen-velocity in the vertical direction [14].	0.68

Table 1: The statistical features used to evaluate the biometric authentication system. Features were chosen based on the quality score Q defined in §4.3. θ is the angle of a given stroke, v, v_x, v_y are overall, horizontal, and vertical velocity, respectively.

How to Build a Low-Cost, Extended-Range RFID Skimmer

Ilan Kirschenbaum[*] *Avishai Wool*[†]

Abstract

Radio-Frequency Identifier (RFID) technology, using the ISO-14443 standard, is becoming increasingly popular, with applications like credit-cards, national-ID cards, E-passports, and physical access control. The security of such applications is clearly critical. A key feature of RFID-based systems is their very short range: Typical systems are designed to operate at a range of 5-10cm. Despite this very short nominal range, Kfir and Wool predicted that a rogue device can communicate with an ISO-14443 RFID tag from a distance of 40-50cm, based on modeling and simulations. Moreover, they claimed that such a device can be made portable, with low power requirements, and can be built very cheaply. Such a device can be used as a stand-alone RFID skimmer, to surreptitiously read the contents of simple RFID tags. The same device can be as the "leech" part of a relay-attack system, by which an attacker can make purchases using a victim's RFID-enhanced credit card—despite any cryptographic protocols that may be used.

In this study we show that the modeling predictions are quite accurate. We show how to build a portable, extended-range RFID skimmer, using only electronics hobbyist supplies and tools. Our skimmer is able to read ISO-14443 tags from a distance of $\approx$ 25cm, uses a lightweight 40cm-diameter copper-tube antenna, is powered by a 12V battery—and requires a budget of $\approx$ \$100. We believe that, with some more effort, we can reach ranges of $\approx$ 35cm, using the same skills, tools, and budget.

We conclude that (a) ISO-14443 RFID tags can be skimmed from a distance that does not require the attacker to touch the victim; (b) Simple RFID tags, that respond to any reader, are immediately vulnerable to skimming; and (c) We are about half-way toward a full-blown implementation of a relay-attack.

[*]Ilan Kirschenbaum is with the School of Electrical Engineering Systems, Tel Aviv University, Ramat Aviv 69978, ISRAEL E-mail: ilankir@gmail.com.

[†]Avishai Wool is with the School of Electrical Engineering Systems, Tel Aviv University, Ramat Aviv 69978, ISRAEL E-mail: yash@acm.org.

1 Introduction

1.1 Background

Radio Frequency Identification (RFID) technology, using the ISO-14443 standard [ISO00], is rapidly becoming widely adopted by many governmental, industrial and commercial bodies. Typical applications include contactless credit-cards, national-ID cards, E-passports, and physical access control (cf. [Fin03], [GSA04]). The security of such applications is clearly critical.

A key security feature of RFID-based systems is their very short range: ISO-14443 systems are designed to operate at a range of 5-10cm. Thus, the perception is that the RFID tag (or smartcard) must almost touch the RFID reader, which should imply that the tag's owner is physically present and holding the tag. Unfortunately, this perception is incorrect. Recently, Kfir and Wool [KW05] described a relay-attack on RFID systems, that violates the implication that the tag being read is in fact near the RFID reader. Their system architecture involves two devices, a "leech" and a "ghost", that communicate with each other (see Figure 1). Such a system would, for instance, allow an attacker to make purchases using a victim's RFID-enhanced credit card—despite any cryptographic protocols that may be used.

As part of their work, [KW05] predicted that the rogue "leech" device can communicate with an ISO-14443 RFID tag from a distance of 40-50cm, based on modeling and simulations. Moreover, they claimed that such a device can be made portable, with low power requirements, and can be built very cheaply. However, beyond acting as a component in a relay-attack, a "leech" can also be used as a stand-alone RFID skimmer, to surreptitiously read the contents of simple RFID tags. Our goal

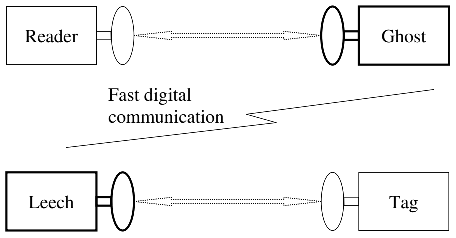


Figure 1: Relay attack system overview.

in this work was to actually build such a skimmer.

1.2 Related work

1.2.1 Attacks on RFID Systems Using the ISO-14443 Standard

The starting point of our work is [KW05]. Their analysis predicts that an RFID tag can be read by the "leech" from a range of tens of centimeters, much further than the nominal ISO-14443 range of 5-10 cm. They also claimed that the "ghost" device can communicate with the reader from distance of tens of meters. [KW05] presented several variants of possible relay-attack implementations, with different costs and required personnel skills. In this work we validate their claims about the practically of the leech device.

Another part of the relay attack against ISO 14443A RFID systems was implemented by Hancke [Han05]: He implemented the fast digital communication between the leech and the ghost (see Figure 1), while using standard (nominal range) devices for the leech and ghost themselves. His system used cheap radios, and achieved a range of 50 meters between the reader+ghost and the leech+tag. His work demonstrates that the range between the victim tag and the reader is limited only by the technology used for leech-ghost communication. To counter the relay attack, [HK05] have designed a distance-bounding protocol, which requires ultra-wide-band communication.

In a widely reported work, Finke and Kelter [FK05]) managed to *eavesdrop* on the communication between an ISO-14443 RFID reader and a tag. They attached the tag directly to a reader (at zero distance), and showed that the combined communication between the reader and tag can be read from 1-2 meters by large loop antenna located on the same plane of the reader and the tag. Note that this is quite different from the challenges facing a skimmer: (a) The skimmer must be close enough to the tag, and produce a strong enough magnetic field, to power the tag (i.e., the tag must be within "activation range"); (b) A skimmer cannot rely on a legitimate reader's strong signal being modulated by the tag. Nevertheless, [FK05] shows that the eavesdropping range on RFID communication is a much greater than skimming range—and we show that skimming range is much greater than the nominal read range.

1.2.2 Attacks on RFID Systems Using Other Standards

There are many RFID systems that do not use the ISO-14443 standard. Typically, such systems are designed for larger read-ranges, but provide much more limited capabilities than ISO-14443: they are unable to power a programmable smartcard processor, and usually only contain fixed logic circuitry or even just a short piece of data, much like a magnetic-stripe card. Over the last 2 years, several attacks have been reported against some of these systems.

In a very widely reported event [Kre05, Sch05], a group from Flexilis claimed to set new world record of passively reading an RFID tag from 69 feet at DefCon'05. However, the RFID technology used for this experiment was not ISO-14443, but a UHF-based technology in the frequency range of 800 MHz to 2.5 GHz which is designed for a much larger read range.

A German hacker ([Hes04]) used a simple PDA, equipped with an RFID read/write device, and changed product prices in a grocery shop using a software he wrote. He managed to reduce the Shampoo price from $7 to $3 and go through the cashier without incident. Supermarket checkout trials held by NCR corporation showed that some clients standing at the cashier paid for groceries held by clients standing behind them in the queue [Whi05].

A research team in Johns Hopkins University ([BGS$^+$05]) managed to build a system that sniffs information from RFID-based car keys and immobilizers, and were able to purchase gasoline without the owners consent.

A research group in MIT ([Lin05]) designed and implemented an RFID field probe that can sense RFID magnetic fields from up to 4 meters. However, it is designed to sense magnetic fields of frequencies between 900 to 950 MHz, which are very different from the 13.56 MHz of the ISO 14443 standard.

1.2.3 RFID Systems and Protocols in General

A broad overview of RFID technology can be found in T.A.Scharfeld's thesis [Sch01]. This thesis analyzes RFID theory, standards, regulations, environment influence, and implementation issues.

Free attack/analysis tools that detect RFID cards and show their meta information are available from the RF-Dump web site [GW04]. These tools are able to display and modify the card data, such as the card ID, card type, manufacturer etc.

Juels, Rivest and Szydlo [JRS03] propose a blocking tag approach that prevents the reader from connecting with the RFID tag. Their method can also be used as malicious tool: In order to disrupt the Reader-to-Tag communication, their blocker tag actually performs a denial-of-service attack against the RFID reader protocol by using the "Tree-Walking Singulation Algorithm" in the anti-collision mechanism. Juels and Brainard [JB04] propose a variant on the blocker concept which involves software modification to achieve a soft blocking tag.

[Wei03] and [SWE02] offer a "Hash-Lock" approach to low cost RFID devices which use a "lock/unlock" mechanism to protect against retrieving the RFID ID number. In the simplest scenario, when the tag is locked it is given a value (or meta-ID) y, and it is only unlocked by presentation of a key value x such that y = h(x) for a standard one-way hash function h.

[RCT05] describe a portable device, called an RFID Guardian, that is supposed to cover a whole individual's surrounding, to communicate with the various tags in the person's possession, and protect the person from potentially hostile RFID fields. The RFID Guardian is supposed to be able to cover a range of 1-2 meters, however, the authors do not describe the RFID Guardian implementation, and it is unclear how it overcomes the physical limitations of the claimed range.

1.3 Contribution

In this study we show that the modeling predictions of [KW05] are quite accurate. We managed to build a portable, extended-range RFID skimmer, using only electronics hobbyist supplies and tools. Our skimmer is able to read ISO-14443 tags from a distance of $\approx$ 25cm, uses a lightweight 40cm-diameter copper-tube antenna, is powered by a 12V battery—and requires a budget of $\approx$ \$100.

Beyond validating the theoretical modeling, we believe that our design, implementation and tuning processes are of independent interest: Most circuit designs and application notes are written for well equipped RF labs, and we needed to modify them or design our own to meet our ridiculously low budget. In particular, our experience shows that the standard RFID tuning process, described in ISO 10373-6 ([ISO01], is inappropriate for hobbyist workshops, and may be missing some key details that are necessary to make it work. Instead, we describe several tuning processes that do work reliably, even in low-budget environments.

We conclude that (a) ISO-14443 RFID tags can be skimmed from a distance that does not require the attacker to touch the victim; (b) Simple RFID tags, that respond to any reader, are immediately vulnerable to skimming; and (c) We are about half-way toward a full-blown implementation of the relay-attack predicted by [KW05].

Organization: Section 2 describes our skimmer system's design. Section 3 describes our construction techniques. Section 4 details the tuning methods we experimented with. Section 5 describes the skimmer's actual performance, and we conclude with Section 6. Additional details can be found in an appendix.

2 System Design

RFID systems that are based on the ISO-14443 standard operate with a 13.56 MHz center frequency, which mandates RF design methods. The system units should be matched for maximal power transfer and efficiency, and the whole system should have an excellent noise figure to improve the receiving and discrimination circuits sensitivity, which in turn allows a large read range.

2.1 Design Paradigms

Our assumption is that we are constructing an ad-hoc system for attack purposes, and mass production is not involved. Therefore modular design and perfect implementation are not the main design goals. Instead, we focused on quick, simple, and cheap methods.

There are two design paradigms that can be followed; the "normal" paradigm is to design all the system subunits to have a uniform 50 Ω input and output impedance. The other paradigm is to design and implement a proprietary RF system, with non-standard characteristics.

The advantages of using standard design include the variety of ready-to-use designs, applications notes, and test equipment. The resulting system is scalable, versatile, and modular. However, the need for accurate design, dealing with accurate filters and semiconductor's min-max parameters and ratings, stretches the design and implementation time, and may cause long and tedious system testing and tuning.

In contrast, designing a proprietary, non-standard interface systems has some practical advantages. First, accuracy is no longer mandatory. Second, the system can work in its natural output and input characteristics without the need to adjust its interfaces to standard characteristics, that might need extra matching networks and components. In particular, some amplifier designs have an output impedance that differs from 50 Ω, and their designated antennas' impedance is closer to the amplifier's impedance than to 50 Ω. In this case, there is no sense to adjust both amplifier output and antenna input to 50 Ω.

Since our goal was to emulate a hacker, we chose to follow the proprietary design paradigm. We used 50 Ω

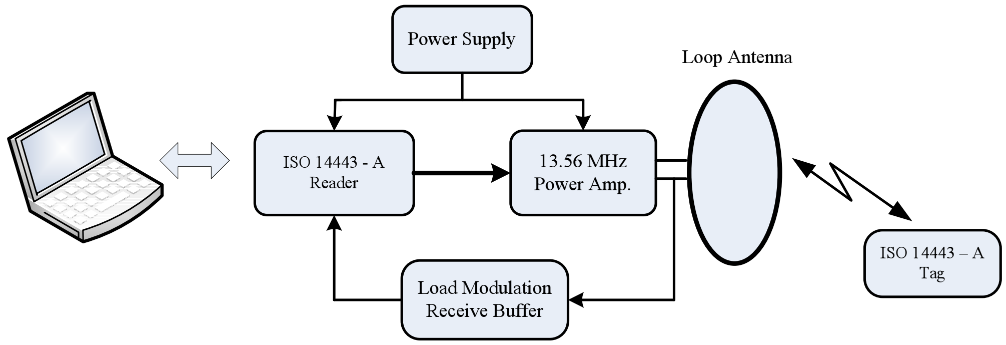


Figure 2: Extended Range RFID Skimmer.

designs where they suited our needs, but we did not attempt to tune all the sub-units precisely. As we shall see, the results were quite satisfactory, despite the very basic work environment and tools.

2.2 System Units

The skimmer is comprised of 5 basic units (see Figure 2): A reader, a power amplifier, a receive buffer, an antenna and a power supply. The RFID reader generates all the necessary RF signals according to the ISO 14443 type A protocol. These signals are amplified by the power amplifier to generate the RF power which is radiated through the loop antenna. The loop antenna performs the interaction with the ISO 14443 RFID tag, and senses the load modulation signals. These signals are buffered by the Load Modulation Receive Buffer and fed back to the reader detection input. The Reader communicates with a host system via an RS232 serial interface. Typically, the host is a computer, however, it can also be a small micro-controller based card, with some non-volatile memory that collects and stores skimmed data.

Our main objective was to increase the output power and antenna size as these two factors directly influence the reading range.

2.3 The RFID Reader

The RFID reader module we used was the Texas Instrument (TI) S4100 Multi-Function reader module, [TI03]. The module can be purchased alone for around $60, and the TI web site ([TI05]) contains sufficient documentation for designing and programming this module. The S4100 module has a built in RF power amplifier that can drive approximately 200 mW into a small antenna. The TI module supports several RFID standards. We focused on the ISO 14443 Type A standard, that is used in contactless smartcards and E-passports.

In addition to the basic S4100 module, we purchased the RX-MFR-RLNK-00 Texas Instrument Multi-Function Reader evaluation kit. The evaluation kit costs $650 and contains a complete ready-to-use reader, which is built around the S4100 module. The kit includes a small built-in 8.5 cm loop antenna and is assembled in a plastic box. It is supplied with basic demo software, various tags for its supported protocols, documentation and references. The kit has an RS232 serial port for interfacing a host computer. We measured a reading range of 6.5 cm using its built in antenna.

Although we could have used the (dismantled) evaluation kit's main board for our experimentation, we chose to build our own base board to demonstrate that buying the evaluation kit is not required. We followed the Interface Circuitry design suggested by TI ([TI03]), but omitted the Low Frequency LED driver. We could have omitted the RS232 level shifters and use TTL levels for the serial port communication, however, the skimmer is supposed to work near the antenna, and to be exposed to strong and noisy electro-magnetic fields, therefore we included common RS232 level shifters in our base board design. This design requires a 5 volts power supply. See Section 2.7 for power supply design and description.

2.4 Antennas

A necessary condition for an increased range is a larger antenna. Theoretical analysis ([Lee03]) shows that for a desired range, $\boldsymbol{r}$, the optimal antenna diameter is $\approx \boldsymbol{r}$. We wanted to demonstrate a reading range of 25-30 cm.

TI's RFID Web site [TI04] supplies an antenna cookbook for building various kinds of antennas for different reading ranges and purposes. As a first experiment, we used a printed PCB 10×15 cm rectangular antenna design found in the cookbook. We later used it as a tuning aid for tuning the system, as described in Section 4. Figure 4 shows the PCB antenna's matching circuit.

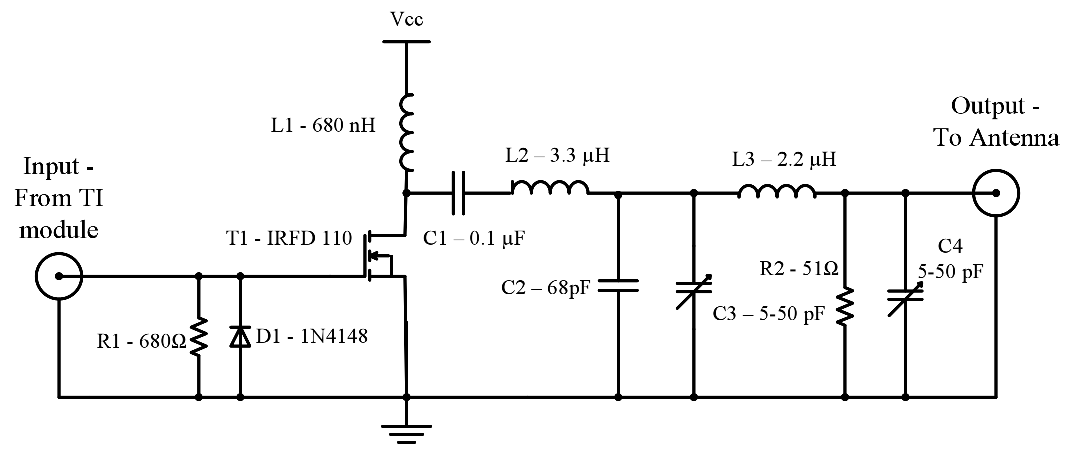


Figure 3: The 13.56 MHz Power Amplifier.

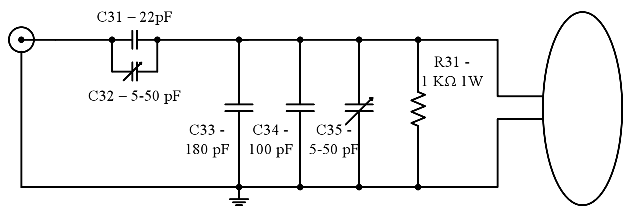


Figure 4: The PCB Antenna 50 Ω Matching Circuit.

For our larger, high power antenna, we constructed a 39 cm copper tube loop antenna. The basic design for the loop antenna's matching network was taken from the PCB antenna (Figure 4) , subject to minor changes: Specifically, the resonance parallel capacitors C33 and C34 that were merged into one capacitor of 82pF, since the calculated antenna's inductance was around 1 μH.

2.5 Power Amplifier

We based our power amplifier on [Mel04], and modified it to suit our unit's interface. The scheme of the power amplifier we designed appears in Figure 3. We interfaced the power amplifier directly to the TI module's output stage embedded in the skimmer base board. However, we did not match impedances between the two since we did not have to transfer power to the power amplifier, but only drive its input for biasing the power FET by a sufficient voltage swing.

2.6 The Load Modulation Receive Buffer

The TI S4100 module is designed around the S6700 Multi-Protocol Transceiver IC, an integrated HF reader system that contains all the high frequency circuitry comprising an Analog Front End (AFE) that decodes the ISO standards protocols. The S6700 has a Receiver input, which is directly connected to the reader's antenna.

This receiver input is unable to handle the voltage levels that are developed on our large loop antenna: During the system development process we measured 184 volts over the antenna with a supply voltage of 17.1 volts. In order to keep the reader from potential damage, and still deliver the load modulation signals to the reader's receiver input, we had to attenuate the antenna signals before feeding them back to the TI module. A simple resistor attenuation network is not suitable since it dramatically influences the antenna's resonance circuit quality factor, Q. Therefore, we chose to use an attenuating buffer (See Figure 5). The buffer was designed using a high impedance RF FET (T2 in Figure 5), in order to keep the antenna's quality factor as designed. The buffer was attached to the antenna and to the TI module via a direct coupling connection, in order to reduce the signal phase shifting to minimum. The C21 variable capacitor is used to compensate for the parasitic capacitance introduced by the T2 FET.

2.7 The Power Supply

In order to drive the large loop antenna, we needed to provide a power supply.

For lab work, we used a stabilized external power supply. Note that the base board that embeds the TI module contains a voltage regulator, therefore the external power supply unit does not have to be regulated. Nevertheless, we used a regulated power supply to reduce its noise figure. Figure 6 shows the regulation and filtering circuity which we placed on the base board and on the power am-

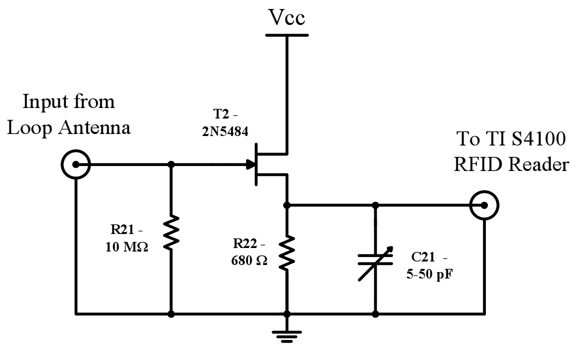


Figure 5: The Load Modulation Receive Path buffer.

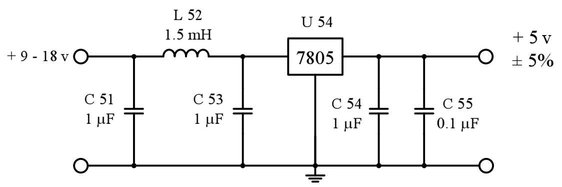


Figure 6: Power supply filter for the reader base board and the power amplifier.

plifier board.

The role of L52 in Figure 6 is to maintain clean and low ripple levels on the DC supply in order to keep a low noise figure of the DC supply voltage. Since the DC supply voltage reaches all the internal chips circuitry, having clean DC voltage to the internal load modulation signals detection circuitry can improve detection range.

To demonstrate the skimmer's mobility, we also operated it using a Non-Spillable 7 AH Zinc-Lead rechargeable battery used in home security systems. It has a 12 volts nominal voltage level, is very common and can be purchased in any home security system store. An added bonus of using a non-switched DC power supply is that it eliminates any switching noise.

3 System Building

3.1 Printing a PCB Antenna

Our first choice was to build a home made 10×15 cm PCB RFID antenna which is fully specified in the TI antenna cookbook. To demonstrate the low-tech requirements, we manufactured this antenna in our hobby workshop. Appendix A describes the PCB printing process.

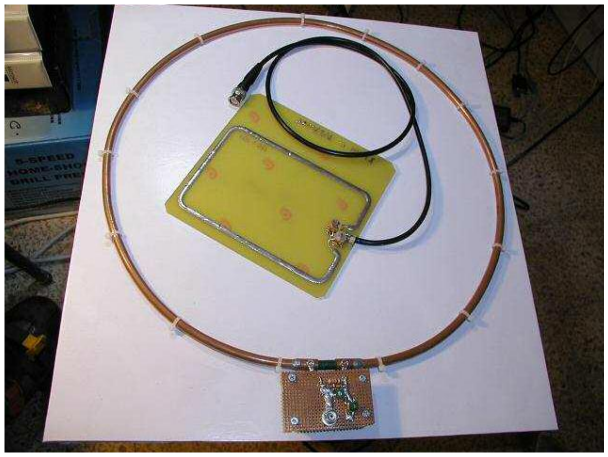


Figure 7: The copper-tube loop antenna and the PCB Antenna.

3.2 Building a Copper Tube Loop Antenna

The TI cookbook describes a design for a square 40×40 cm copper-tube antenna, which seemed appropriate. However, we chose not to construct it precisely, since cheap copper tube (for cooking gas) is sold packed in circular coils, and constructing an antenna with a square or rectangle shape requires straightening the tube, and requires additional 90 degrees matching adapters, which increase the antenna's cost. Instead, we designed our own circular antenna, which has similar characteristics to the TI cookbook antenna.

We built the loop antenna from 5/16 inch cooking gas copper tube. The tube is tied to a solid non flexible wooden tablet, in order to maintain its shape and to avoid inductance changes under mechanical deformation forces.

The loop antenna construction process was basically mechanical handcraft work, requiring no special equipment beyond basic amateur's electrical tools. Note that copper tube must be soldered using at least a 100-watts blow torch. Figure 7 shows the finished copper tube antenna and the PCB antenna.

3.3 Building the RFID Base Board

According to the interfacing information we found in the S4100 module datasheet, we designed a small PCB base board, having the S4100 module as a Piggy Back.

We manufactured the RFID base board PCB using a different method than we used to make the PCB antenna. For this board, we used a Decon DALO 33 Blue PC Etch protected ink pen to draw the leads on the Glass-Epoxy tablet. This technique allowed us to print the PCB during any time of day, without the need to wait for the sun. See

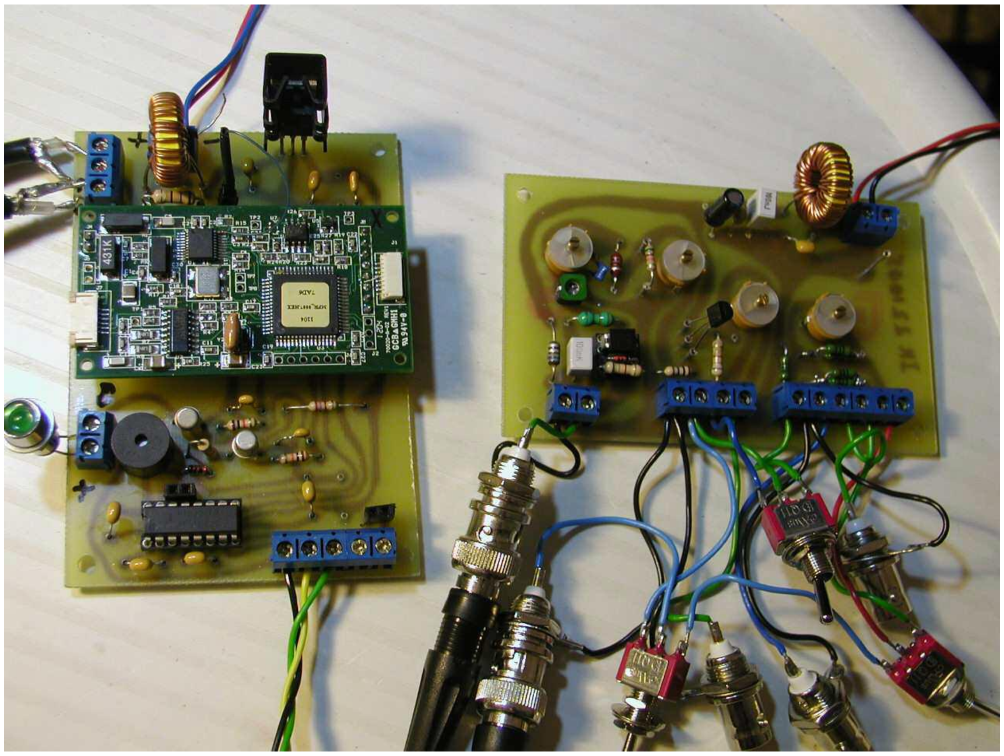


Figure 8: The TI S4100 module mounted on our base board on the left, and the power amplifier board on the right.

Figure 8 for a picture of the base board.

3.4 Building the Power Amplifier

As we noted in Section 2.5, the power amplifier design is based on a Melexis application note ([Mel04]), recall Figure 3. We used the output stage of the TI S4100 reader module in the base board to drive the power amplifier input. We did not invest any effort in impedance matching since the power amplifier input is voltage driven. We manufactured the PCB for the power amplifier using the same technique as used for the base board, and with the same low cost DC ripple filter (recall Figure 6) to maintain a low noise figure.

Beyond the Melexis design, empirical results led us to connect a filter comprised of R2 and C4 at the output (See Figure 3). This filter reduces the Q of the output impedance matching filter, enabling fine tuning of the output signal phase. We discovered that the filter increased the read range significantly.

The output voltage amplitude of the power amplifier varies depending on the power supply voltage. For instance, with a 17.1 volts power supply we measured over 180 volts on the resonance circuit and the antenna. Therefore, ideally, high voltage rating capacitors, and high current rating inductors should be used. We used regular, but easy to obtain, passive components, and managed to burn quite a few during our experimentation.

3.5 Building the Load Modulation Receive Path Buffer

As we mentioned before, the high voltage swing on an antenna driven by the power amplifier must be attenuated in order to supply the correct samples of the RF received signal back to receive input of the S4100 module. Therefore, we needed to build the buffer described in Section 2.6. We placed the buffer's circuitry on the same PCB that housed the power amplifier - see Figure 8.

One challenge we had to deal with is that the TI S4100 module is designed to work with a low power antenna, and includes an attenuation resistor that is suitable for such an antenna. In order to provide our (attenuated) signals to the S4100, we had to solder the buffer's output directly into the S4100 module, bypassing the original attenuation resistor. Figure 9 shows the bypass.

4 System Tuning

A crucial implementation phase is system tuning and adjustment. Specifically, we have to tune the various reso-

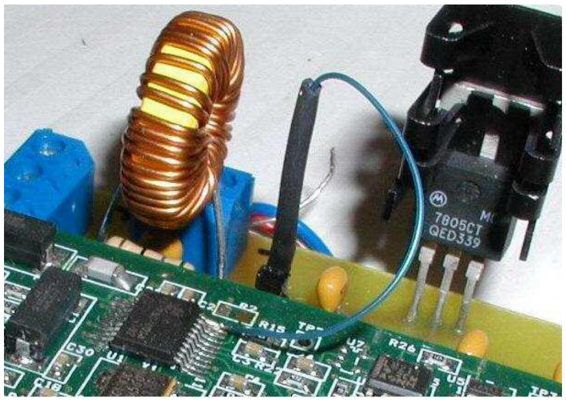


Figure 9: The direct connection to the TI module, bypassing the existing attenuation resistor.

nance circuits and matching networks for maximal power transfer. The only test equipment we used throughout the entire project was cheap 60 MHz oscilloscope, that any electronic hobbyist has in his workshop. Note that while resonance frequency can be tuned using an oscilloscope, matching the antenna to the amplifier requires a different procedure since both a magnitude and a phase must be matched.

4.1 Standard Tuning Methods

We say that a tuning method is "standard" if it requires a 50 Ω design.

The first and most straightforward tuning method is to use an RF network analyzer. Among its various features, a network analyzer can measure the magnitude and phase of a system input, allowing us to know exactly what matching network to connect to this system in order to match it to the desired impedance. In our case, a network analyzer can measure the antenna input impedance, e.g., its phase and magnitude, which would enable us to calculate the matching circuitry for 50 Ω input impedance. In case we already have a matching network, the RF network analyzer can measure the return loss and let us tune the system to minimum returned power. Unfortunately, an RF network analyzer costs over $10,000, well beyond the budget of an amateur.

Another tuning method is to measure the Voltage Standing Wave Ratio (VSWR), and to adjust the antenna's impedance to be best matched to the driving amplifier output stage by tuning the returned power to the desired value ([Poz05]). This method requires a VSWR meter, which costs several hundred US$: still beyond a typical hobbyist budget. A cheap way to measure the VSWR (without a VSWR meter) is to use directional couplers, that cost between $20–$70, but their input and output impedance is 50 Ω, requiring 50 Ω interface subsystems design. We have not attempted this method.

Finally, one can tune the system using an RF wattmeter, or an RF power meter. These instruments sense the RF power and translate the sensor's measurement to a standard scale. The sensor can be based on a diode, or on a bolometer: an RF power sensor whose operation is based on sensing purely resistive element radiation. This method is a second-order-effect tuning since it measures the antenna power reception rather than the actual direct amplifier to antenna matching. This kind of equipment costs between $300 (used) to $600 for a simple watt-meter, including the sensor, to about $3000 for an RF power meter that also features a VSWR meter and various other RF measurement capabilities.

4.1.1 The ISO 10373 Tuning Method

Since tuning the RFID receiver is a critical part of building such a device, Annex B of the ISO 10373-6 standard ([ISO01]) suggests a tuning process. This process seemed attractive since it only calls for low-cost components and uses basic oscilloscope capabilities. Therefore, despite the fact that ISO 10373 is a standard (50 Ω) tuning process, we invested a significant effort into trying to use it. Our experience leads us to conclude that the process is not very effective, at least for hobbyist setups.

The ISO 10373-6 testing configuration is based on monitoring a phase difference between the signal source and the load. The monitoring device utilizes a standard oscilloscope for displaying Lissajous figures in XY display mode, see Figure 10. If the time constant of the reference network equals the time constant of the network formed by the calibration resistor along with the oscilloscope Y probe's parasitic capacitance, no phase difference should be monitored. If there is a difference in the two time constants, there will be a phase shift between the two probes of the oscilloscope, and the Lissajous figure should form an ellipse, whose main axis is at a 45-degree angle. The "fatness" of ellipse is related to the phase difference: when the system is perfectly tuned, there is no phase difference, and the Lissajous figure collapses to a straight line.

[ISO01] has two steps. The first step calibrates the test set to eliminate the oscilloscope input impedance from influencing the tuning step. In this step, the impedance matching network and the antenna of Figure 10 are replaced with a 50 Ω resistor to simulate 50 Ω load. The second step is the actual antenna tuning step. In this step, we replace the calibration resistor with the antenna containing the matching circuit, and trim the capacitors until we monitor that the Lissajous figure is closed, indicating a zero phase shift.

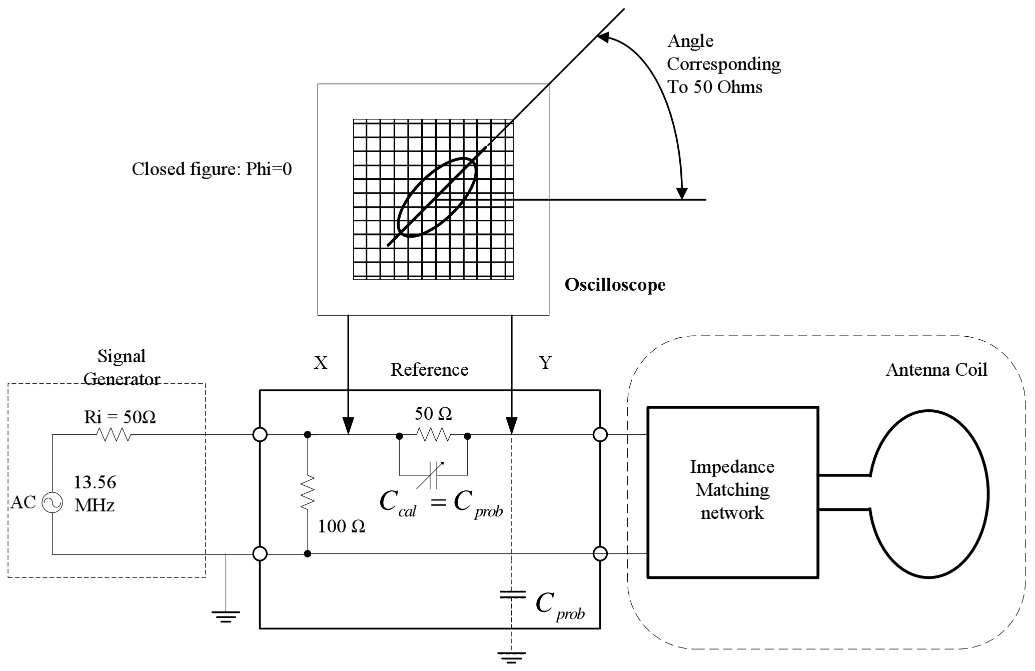


Figure 10: The ISO-10373 setting—Matching the antenna input impedance to 50 Ω. In the first step, the impedance matching network and the antenna are replaced with 50 Ω resistor to simulate 50 Ω load.

4.1.2 Problems with ISO 10373

Despite its apparent simplicity, in practice we discovered that the ISO 10373-6 tuning process has a few problems.

The first thing to note is that this tuning method requires a 13.56 MHz signal source, with a 50 Ω output impedance, that can deliver enough power to drive an antenna such as our copper tube antenna. We invested a significant effort trying to build a clean and accurate signal source, but this turned out to be difficult to do in reasonable time. Even obtaining an accurate 13.56 MHz crystal proved to be problematic—none of the electronics suppliers we contacted carried such a component. To bypass this obstacle, we decided to use the TI S4100 module itself as a signal source—since it is matched to 50 Ω and can drive sufficient power to the antenna. Once we did this, we were able to construct the rest of the circuitry, and we tried to tune the antenna.

Unfortunately, in all possible settings of the antenna's matching circuitry, we did not manage to get the expected Lissajous figures. In some settings we got wavy figures implying a non linear circuit working point. In other settings we did not get the figures centered around the desired 45 degrees slope. Worst of all, we found no correlation between more closed Lissajous figures and longer read ranges (which we obtained using the methods of sections 4.2 and 4.3.

To our frustration, we found that ISO 10373-6 does not specify the exact oscilloscope Volt per Division setting. This level of detail matters since we are dealing with very fine tuning, and human eye, oscilloscope line thickness and human judgment in conjunction with parallax error, can lead to errors. We speculate that if major RF labs indeed use this standard for tuning, they probably have some additional "secret sauce" that fills in the missing details.

One possible reason for our difficulties may be that we used the TI module as a signal source. This improvisation may have inserted some undesired harmonics due to the sidebands in the downlink signal spectrum, interfering with the tuning process. Since the methods described in Section 4.3 were effective, we did not pursue this further.

4.2 Non-Standard Tuning Methods

Instead of the standard 50 Ω tuning methods, we used the following two non-standard methods. We found that they both work reliably, and give satisfactory results.

One tuning method includes sensing the reception power using a small loop antenna as a sensor, leading its receptive power to a home-made RF power meter. The RF power meter can be an AM demodulator, whose DC level is proportional to the received RF power, or a home-made bolometer—we chose to use the latter.

The other non-standard method is a trial-and-error iterative process of reading an RFID tag at increasing distances, while tuning the matching circuitry, until a maximal range is reached.

4.3 Tuning Methods that Worked

The antenna has two tuning steps. The first is tuning the resonance frequency by trimming capacitor C35 in Figure 4. The second step is tuning the series capacitor C32 in Figure 4 to achieve maximal power transfer to the antenna. For tuning the resonance circuit we used the power amplifier's output signal, driven by the reader base board to tune center resonance frequency.

Then, for tuning the entire system, we used the iterations method described earlier. For this we used a Philips Mifare Standard IC tag. Initially, we located the tag at the basic range according to the RFID standard, and tuned the series antenna network capacitor C32 to some initial tag read. When an initial reading is observed, we know that the final position of C32 is near the position of the initial readout. We gradually increase the power supply, and each time adjust the various capacitors to get a stable reading range, while increasing the distance between the tag and the antenna. To hold the tag at a fixed distance we used non-ferromagnetic objects: most of the time we used a stack of disposable plastic cups, and for fine range measurements we used a small supply of 1-2mm thick beer coasters, see Figure 12. We stopped at a 19 volts power supply since the maximum semiconductor ratings were reached. Surprisingly, the variable capacitors survived the high swing voltage, which was more than 180 Volts.

During the iterations, a secondary source of tuning information was the sound level from the computer speakers. We turned the speakers to their maximum volume while we tuned the antenna matching capacitor: The tuning process caused the speakers to hum, and their loudness gave an idea how close we are to the final matching.

One disadvantage of this iterative method is its sensitivity to different tags: Some tags gave larger read ranges than others. On the other hand, the process is simple and quick: It took us approximately 10 minutes to tune the system to maximal performance.

A second tuning method that worked was based on a bolometer. We placed our smaller PCB antenna in the magnetic field produced by the large loop antenna, and measured changes in the RF power it was exposed to. Instead of purchasing an expensive industrial RF wattmeter or bolometer, we built our own: We attached a 100 KΩ thermistor to a 50 Ω resistor using super glue. To improve the bolometer performance, we increased its thermal conductivity by using a silicon thermal grease around the attaching surface between the resistor and the thermistor, see Figure 11. To keep it isolated from ambient temperature, we then covered it with a small piece of isolating PVC sleeve, used for thermal isolation of copper hot water pipes. Note that our bolometer is not calibrated to any standard units — but this is unimportant since all we care about is to reach a maximum value; we do not need to quantify the level of received RF power.

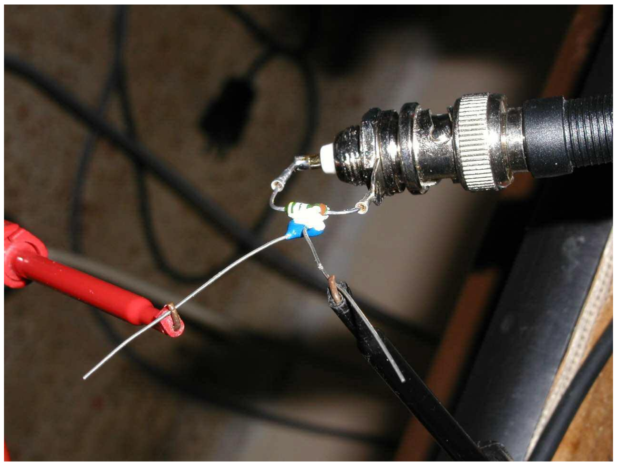


Figure 11: Home made bolometer using a resistor and a glued thermistor.

Using a binary search, while examining the amplitude over the antenna and reading the bolometer's resistance-temperature, we tuned the matching capacitor until we observed the maximum temperature. The results were accurate, and we reached the same final position of C32 that we marked at the end of iterations tuning process. This tuning method is independent of a particular tag—but it is slower, since it takes $\approx$ 15 seconds per setting for the thermistor to adjust to a new temperature and for the bolometer's reading to stabilize.

4.4 Miscellaneous Tuning Tips

4.4.1 Strong Magnetic Fields

Note that the antenna's magnetic field is so strong that it crashed one of the lab's computers even though it was approximately 1 meter away. We had to format the disk and re-install the OS and all applications.

4.4.2 Power Amplifier Tuning

The power amplifier has a simple tuning procedure. First, position the the C3 capacitor at its mid-point, and get a first readout from the tag. Then tune the antenna as described before. Finally, after tuning the antenna to maximal power matching, fine-tune C3 and attempt to increase the read range further.

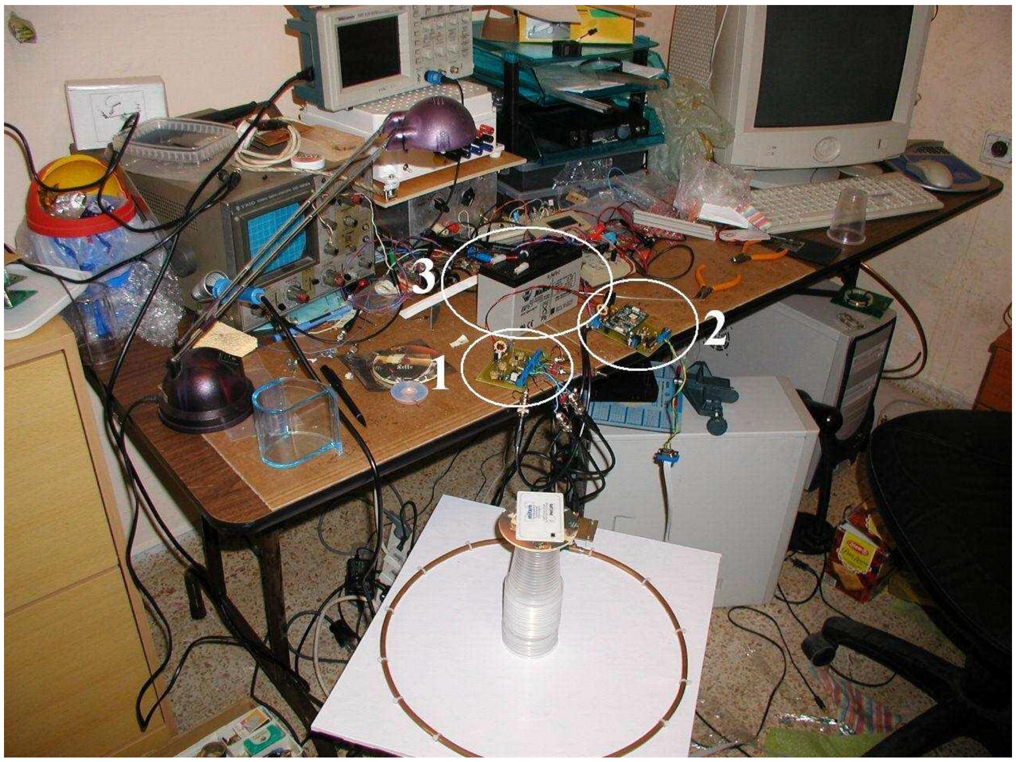


Figure 12: The antenna tuning process. Note the tag placed over a stack of plastic cups and beer coasters in the center of the antenna. The power amplifier is marked as item 1, the reader base board is marked as item 2 and the battery is marked as item 3.

4.4.3 The Effect of a Battery Power Supply

During our lab work we used a linear stabilized power supply. We assumed that once we attach our system to a battery the reading range will grow because the battery delivers clean and ripple free voltage. However, in practice, we got only few millimeters improvement, if any. We believe that our linear power supply has quite a low noise figure so it gave us similar ranges to those achieved using a battery.

4.4.4 Surrounding Metal Objects

While tuning the antenna, care should be taken to remove any metal objects near the antenna. Reflections, grounded metal surfaces, and metal object permeability can influence the antenna's magnetic fields, leading to erroneous results. Even the human hands can influence the tuning results. To overcome these kinds of problems, we used only non ferromagnetic accessories, like a plastic table for laying the antenna, a wooden stick with the RFID tag attached to its edge for coarse range measurement, and plastic cups and coasters for fine range measurement.

5 Results

5.1 Achieved Read Ranges

Our reference system was the RX-MFR-RLNK-00 Texas Instrument Multi-Function Reader evaluation kit. The evaluation kit embeds the TI module we used, and comes with small 8.5 cm diameter round antenna directly connected to the module's output [TI05]. The basic read-range of the evaluation kit was 6.5 cm.

We first connected our 10×15 cm PCB antenna to the evaluation kit, without the power amplifier. This alone gave a range increase of 30%, to around 8.5 cm. Attaching the big loop antenna to the evaluation kit gave no results since the kit generates only 200 mW output power that is insufficient to drive the antenna.

Using the power amplifier we reached much larger ranges (see Figure 13). With the linear power supply providing 14.58 volts, we were able to read the tag at a range of 17.3 cm using the PCB antenna, and at a range of 25.2 cm using the copper tube antenna. With a 12-volt battery we reached a reading range of 23.2 cm using the copper tube antenna and 16.9 cm using the PCB antenna. Note that this type of battery, upon recharging, can sup-

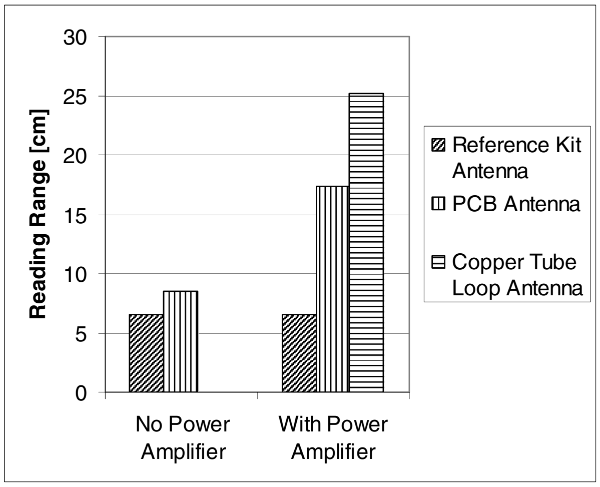


Figure 13: Skimmer read-range results with the reference kit antenna, PCB antenna, and the copper tube loop antenna, with and without the power amplifier.

ply more than its nominal voltage: we measured that it supplied 13.8 volts during the above experiments.

We observed that increasing the power supply voltage did not always cause a range increase: Higher power levels sometimes caused lower reception. This is in line with the predictions of [KW05]. The reason is that the distortion inserts harmonics that interfere with the detection of the side bands that are about 60 to 80 dB under the 13.56 MHz carrier power. We found that the optimal power supply voltage for our antennas was around 14.6 volts.

5.2 Comparison with the Theoretical Predictions

We measured a 170mA DC supply current to the power amplifier when using the the copper tube antenna. The combination of this current value and a read range of 25 cm match the predictions of [KW05] very well: The graph shown in Figure 14 is from [KW05], and the star indicates our empirical results on the predicted curve.

We believe that using high rating components and more powerful RF test equipment, we can reach the road map along the theoretical curve. This will be done in later work.

5.3 System Cost

Ignoring the time and cost of labor, the system cost is ridiculously low. The most expensive item in the system is the TI module, which costs around \$60. All the other components, the materials for the PCBs, and the items needed for building the loop antenna, together cost at most \$40-\$50, giving a total cost of \$100-\$110.

6 Conclusions

In this work we have shown how to build a portable, extended-range RFID skimmer. Our skimmer is able to read ISO-14443 tags from a distance of $\approx$ 25cm, uses a lightweight 40cm-diameter copper-tube antenna, is powered by a 12V battery—and requires a budget of $\approx$ \$100. We were able to build and tune the skimmer using only electronics hobbyist supplies and tools. By doing this we have proved three things: First, we have validated the basic concept of an RFID “Leech” and the modeling and simulation work of [KW05]. Second, we have demonstrated that ISO-14443 RFID tags can be skimmed from a a range that is 3-5 time larger than the nominal range, and more importantly, is a distance that does not require the attacker to touch the victim. This last observation can make a noticeable difference in the attacker’s mode of operation. Finally, we are about half-way toward a full-blown implementation of a relay-attack of [KW05].

Our work implies that simple RFID tags, that respond to any reader, are immediately vulnerable to skimming. Therefore, at the very least, RFID tags, and in particular E-passports, should incorporate additional controls that prevent the tag from being read surreptitiously: e.g., physical shielding inside a Faraday cage, and cryptographic application-level access controls that require the reader to authenticate itself to the tag.

However, in isolation, cryptographic controls can only protect against skimming—they cannot protect against a relay attack. To protect against a relay attack, the RFID tag must be equipped with additional physical controls such as an actuator, or an optical barcode physically printed on the passport jacket: these help ensure that the reader is in fact reading the tag that is presented to it and not some remote victim tag.

Acknowledgments

We would like to thank Sergey Belous, Sammy Datika, Klaus Finkenzeller, Yeheal Greenblat, Ziv Kfir, Motti Kirshenboim, Markus Kuhn, and Moshe Panijel, for many stimulating discussions and practical tips, that greatly helped us during this project.

References

[BGS+05] S. Bono, M. Green, A. Stubblefield, A. Juels, A. Rubin, and M. Szydlo. Security analysis of a cryptographically-enabled RFID device. `http://rfid-analysis.org/DSTbreak.pdf`, 2005.

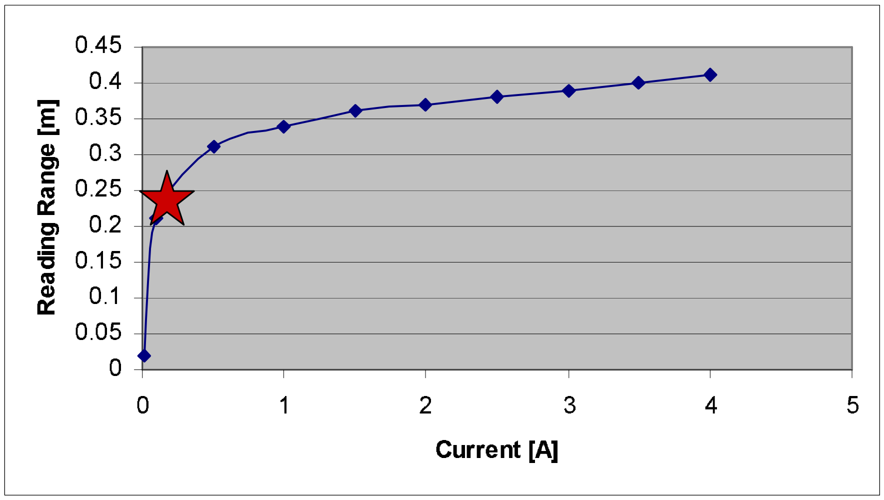


Figure 14: The predicted read-range versus the antenna current from [KW05]. The star indicates our empirical results.

[Fin03] Klaus Finkenzeller. *RFID Handbook: Fundamentals and Applications in Contactless Smart Cards and Identification*. John Wiley & Sons, 2003.

[FK05] Thomas Finke and Harald Kelter. Radio frequency identification— Abhörmöglichkeiten der Kommunikation zwischen Lesegerät und Transponder am Beispiel eines ISO14443-Systems. BSI - German Ministry of Security, 2005. `http://www.bsi.de/fachthem/rfid/Abh_RFID.pdf`, in German.

[GSA04] U.S. government smart card handbook. Office of Governmentwide Policy, General Services Administration, February 2004.

[GW04] Lukas Grunwald and Boris Wolf. RFDump, 2004. `http://www.rf-dump.org/`.

[Han05] Gerhard Hancke. A practical relay attack on ISO 14443 proximity cards, 2005. `http://www.cl.cam.ac.uk/~gh275/relay.pdf`.

[Hes04] Arik Hesseldahl. A hacker's guide to RFID. *Forbes Electronic Magazine*, July 29 2004. `http://www.forbes.com/home/commerce/2004/07/29/cx_ah_0729rfid.html`.

[HK05] Gerhard Hancke and Markus Kuhn. An RFID distance bounding protocol. In *Proc. 1st International Conference on Security and Privacy for Emerging Areas in Communication Networks (SecureComm)*, Athens, Greece, September 2005. IEEE.

[ISO00] Identification cards – contactless integrated circuit(s) cards – proximity cards - part 1 to 4. ISO/IEC 14443, 2000.

[ISO01] Identification cards – test methods – proximity cards - part 6, annex B. ISO/IEC 10373-6, 2001.

[JB04] A. Juels and J. Brainard. Soft blocking: Flexible blocker tags on the cheap, April 2004. `http://theory.lcs.mit.edu/~rivest/`.

[JRS03] A. Juels, R. Rivest, and M. Szydlo. The blocker tag: Selective blocking of RFID tags for consumer privacy. In *Proc. 8th ACM Conf. Computer and Communications Security (CCS)*, pages 103–111, May 2003. `http://theory.lcs.mit.edu/~rivest/`.

[Kre05] Brian Krebs. Leaving Las Vegas: So long DefCon and Blackhat. The Washington Post, 2005. `http://blogs.washingtonpost.com/securityfix/2005/08/both_black_hat_.html`.

[KW05] Ziv Kfir and Avishai Wool. Picking virtual pockets using relay attacks on contactless smartcard systems. In *Proc. 1st International Conference on Security and Privacy for Emerging Areas in Communication Networks (SecureComm)*, pages 47–58, Athens, Greece, September 2005. IEEE.

[Lee03] Youbok Lee. Antenna circuit design for RFID application. Microchip Technology, Application Note AN710, DS00710C, 2003. `http://ww1.microchip.com/downloads/en/AppNotes/00710c.pdf`.

[Lin05] Rick Lingle. MIT's economical RFID field probe, 2005. `http://www.packworld.com/articles/Departments/18784.html`.

[Mel04] A power booster for the MLX90121. Melexis Application Note 390119012102, Rev.001, April 2004. `http://www.melexis.com/relinfofiles/AN90121_1.pdf`.

[Poz05] David M. Pozar. *Microwave Engineering*. John Wiley & Sons, Inc., third edition, 2005.

[RCT05] Melanie Rieback, Bruno Crispo, and Andrew Tanenbaum. RFID guardian: A battery-powered mobile device for RFID privacy management. In *Australasian Conference on Information Security* *and Privacy – ACISP'05, LNCS 3574*, pages 184–194, Brisbane, Australia, July 2005. Springer-Verlag.

[Sch01] Tom A. Scharfeld. An Analysis of the Fundamental Constraints on Low Cost Passive Radio-Frequency Identification System Design. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA 02139, August 2001.

[Sch05] Bruce Schneier. RFID passport security revisited. Schneier on Security: A weblog covering security and security technology, 2005. `http://www.schneier.com/blog/archives/2005/08/rfid_passport_s_1.html`.

[SWE02] Sanjay E. Sarma, Stephen A. Weis, and Daniel W. Engels. RFID Systems and Security and Privacy Implications. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS 2523*, pages 454–470. Springer-Verlag, 2002.

[TI03] S4100 multi-function reader module data sheet. Texas Instruments, Module 11-06-22-715, 2003. `http://www.ti.com/rfid/docs/manuals/refmanuals/rf-mgr-mnmn_ds.pdf`.

[TI04] HF antenna cookbook. Technical Application Report 11-08-26-001, Texas Instruments, January 2004. `http://www.ti-rfid.com`.

[TI05] Rfid homepage. Texas Instruments, 2005. `http://www.ti-rfid.com`.

[Wei03] Stephen A. Weis. Security and Privacy in Radio-Frequency Identification Devices. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA 02139, May 2003.

[Whi05] Dan White. NCR: RFID in retail. In S. Garfinkel and B. Rosenberg, editors, *RFID: Applications, Security, and Privacy*, pages 381–395. Addison-Wesley, 2005.

A Printing the PCB antenna

The PCB antenna was made using PCB printing materials and hobbyist equipment as listed below:

- Raw PCB Glass-Epoxy tablet size 20x25 cm - price $5
- Photo resist, Positive process - $27
- Ferric Chloride - $9
- Soda Caustic - $9
- Piece of glass, size 18x23 cm for standard photo frame - $1
- 1 A4 Pergament paper - 20 cents
- Black alcohol based non erasable water proof pen - $1.25
- Acetone - $4
- Rubber gloves, can be bought in a Dollar Store - $1

The cookbook contains a complete description, including a print layout and electronic circuitry (See [TI04] pages 21-22).

The process of making the PCB antenna is identical to the process of making any PCB. Note that positive photo-resist PCB printing requires a positive layout film. Since making a celluloid film requires photographic equipment, we used the more common materials.

We first printed the antenna PCB layout on the pergament paper using an ink injection printer set up as follows:

- Print quality - best paper setting
- Transparency film - other transparency film
- Color - Print in gray scale - black only
- Check the GUI check box for "Actual size"
- Ink Volume - Heavy

The following instructions guide you through the antenna manufacturing process. Wear rubber gloves and protect eye glasses since Ferric Chloride acid is a very strong and harmful material, and contact with human eyes causes severe injury.

1. Cover the large areas of the ink with the water proof pen to avoid any penetrating light through the pergament paper.
2. Prepare the raw PCB Glass-Epoxy tablet for exposure by thoroughly cleaning it from dust and dirt. A clean surface is crucial to avoid PCB printing flaws.
3. Dry the tablet in an oven at a temperature around 70 Celsius degrees.
4. Thoroughly clean the glass against spots and dust.
5. In a dark room, spray a thin layer of Positive Photo Resist on the PCB tablet, and dry it in the oven at 70 Celsius degrees for about 20 minutes.
6. Make a 7% Soda Caustic solution with water.
7. Put the pergament printed layout over the PCB tablet in the correct direction (be aware of the Print Side (PS) and Component Side (CS)) .
8. Put the glass on the pergament paper and hold them together tightly.

9. Expose the “sandwich” to bright sunlight for 4 to 6 minutes.

10. Remove the glass and pergament paper, and insert the exposed PCB into the Soda Caustic solution for about 20 minutes until all the photo-resist that was exposed to the sun is removed.

11. Thoroughly wash the PCB with water. Be extra careful not to scratch the photo-resist printed leads.

12. Make a 25 Celsius degrees Ferric Chloride solution, and insert the PCB until the exposed copper is fully etched. The PCB should be rapidly shaken within the acid, otherwise the etching process will take a long time. Shaking it will shorten the etching process to around 45 minutes. An aquarium pump is an effective and cheap way to stir the acid.

13. Wash the PCB thoroughly with water, dry it, and use the Acetone to remove the photo-resist from the antenna’s copper leads. We still had few small flaws left due to strong etching, therefore we covered the whole antennas copper leads with tin.

The 50Ω impedance matching network were soldered according to TI Antenna cookbook, see Figure 4, and we used a BNC connectors instead of SMA to reduce cost. At this point, the antenna is ready for tuning and use.

In countries lacking sunny days for long months, one can consider screen printing technology for printing the PCB antenna. This technique requires some background knowledge and some practical experience. The basic materials costs around $150 dollars, and after few attempts, an average handyman can handle the task quite easily. We have not tried the screen printing as the process we described worked successfully for us.

Keyboards and Covert Channels

Gaurav Shah, Andres Molina and Matt Blaze
Department of Computer and Information Science
University of Pennsylvania
{gauravsh, andresmf, blaze}@cis.upenn.edu

Abstract

This paper introduces *JitterBugs*, a class of inline interception mechanisms that covertly transmit data by perturbing the timing of input events likely to affect externally observable network traffic. JitterBugs positioned at input devices deep within the trusted environment (e.g., hidden in cables or connectors) can leak sensitive data without compromising the host or its software. In particular, we show a practical *Keyboard JitterBug* that solves the data exfiltration problem for keystroke loggers by leaking captured passwords through small variations in the precise times at which keyboard events are delivered to the host. Whenever an interactive communication application (such as SSH, Telnet, instant messaging, etc) is running, a receiver monitoring the host's network traffic can recover the leaked data, even when the session or link is encrypted. Our experiments suggest that simple Keyboard JitterBugs can be a practical technique for capturing and exfiltrating typed secrets under conventional OSes and interactive network applications, even when the receiver is many hops away on the Internet.

1 Introduction

Covert channels are an important theoretical construction for the analysis of information security, but they are not often regarded as a significant threat in conventional (non-MLS) networked computing systems. A covert channel allows an attacker that has compromised a secure system component to leak sensitive information without establishing its own explicit connection to the outside world. Covert timing channels, for example, may exist if there is flexibility in the timing or sequencing of externally observable events (such as disk accesses or delivery of data packets). Covert channels are notoriously hard to detect or eliminate, but this is somewhat ameliorated by the fact that their bandwidth is often rather low, and, in any case, exploiting them requires that the attacker somehow compromise a sensitive system component in the first place. The sensitive system component typically gives the attacker total control over the system or an output channel, making the threat of covert channels relatively minor compared with that of whatever software vulnerability which made such a compromise possible in the first place. Outside of those intended explicitly to support multi-level security, conventional general purpose commercial operating systems, network components, application software, and system architectures largely ignore the threat of covert channels.

In this paper, however, we suggest that typical general purpose computing systems are indeed susceptible in practice to certain covert timing channels. These channels require only the compromise of an input channel or device and can leak sensitive information (such as typed passwords and encryption keys) through the network interface. Furthermore, this can remain a threat even under conditions that intuitively seem quite unfavorable to the attacker, where there is only an indirect, multi-stage link between the compromised system component and a receiver placed many hops away on the Internet.

Specifically, we investigate *loosely-coupled network timing channels,* in which a compromised input device is separated from a covert receiver by multiple system layers at different levels of abstraction. Each of these layers adds noise to the timing of received events through normal internal propagation delays, event scheduling, and buffering. The receiver is assumed only to passively measure the arrival times of some subset of network packets but otherwise has no access to sensitive data. We introduce *JitterBugs,* a class of mechanisms that exploit such channels. A JitterBug must have access to sensitive information along with the capability to modulate event timing. JitterBugs can thus capture and store this sensitive information and send it later through a loosely-coupled network timing channel. Loosely-coupled timing channels and JitterBugs provide a practical framework for exploiting timing channels that exist in general-purpose computing systems.

In particular, we built a hardware keylogger, the *Keyboard JitterBug,* that can leak typed passwords over the Internet without compromising the host or its OS, without the use of a separate communication channel, and without the need for subsequent access to the device by the attacker. The Keyboard JitterBug is intended as an interesting artifact in its own right (demonstrating a practical attack tool that can operate under highly constrained conditions), but also as a platform for studying the propagation of timing information across hardware, operating systems, network stacks and the Internet. Assuming that the user is running an interactive network application (e.g. SSH, X-Windows), it can leak previously captured passphrases over such network connections. We show using experiments that one can get good performance independent of the OS, system and network conditions under which the Keyboard JitterBug is deployed. Such a device is therefore very robust against any changes in its environment. Keyboard JitterBugs also raise the threat of a *Supply Chain Attack*. In this attack, a powerful adversary subverts a large number of keyboards in the hope that a target of interest acquires one.

2 Related Work

A common simplifying assumption in the covert channel literature is that the attacker has direct control over the timing of the events being measured by the receiver. That is, the attacker is usually assumed to compromise important system components that allow partial or total access to the output subsystem. While this may be a useful conservative assumption for those concerned with minimizing covert channel bandwidth or for abstractly modeling information leakage, we note that those seeking to *exploit* a timing channel may be able to do so more indirectly. In particular, network packet timing is influenced by many system components outside a host's network subsystem, including input devices. Event timing information is propagated from one layer to another, eventually reaching the external network, where it can be measured by an adversary. We are not the first to observe that packet timing can leak sensitive information about non-network subsystems, which has been effectively exploited in remote timing "side channel" attacks against crypto systems [10] and for host fingerprinting [26, 8, 9]. Here, however, we are concerned not with incidental side channel leakage, but with leakage deliberately introduced (perhaps at somewhat higher bandwidth) by a malicious adversary.

The term "covert channel" was first used by Lampson [27] in describing program confinement to ensure processes were not able to leak private data to other processes. Covert channels have primarily been studied in the context of multi-level secure (MLS) systems. MLS systems have multiple security clearance levels. A "HIGH" level should be able to access any data at "LOW" level but not vice-versa. It is thus important that there be no covert channels that allow a rogue agent (e.g. software trojan horse, spy) to transfer information from "HIGH" to "LOW". As a result, some of the earliest research in covert channels was from the perspective of these systems. Due to resource sharing and some commonly used MLS primitives, getting rid of covert channels in such systems is often very hard and in some cases, effectively impractical [38, 33].

Identification of covert timing channels is concerned with enumerating all possible covert channels that might be exploited by a software or the user. The US Trusted Computer System Evaluation Criteria [2] requires explicit covert channel identification in any system certified at class B3 or higher. Many methods have been proposed to identify covert channels, e.g. dual-clock analysis [46], shared resource matrix [24], high-level scenarios [17]. Note that none of these methods guarantee that all covert channels will be found, and, more importantly, identified channels may represent an exploitable threat.

Once practical covert timing channels have been identified, it is often necessary to take steps to mitigate them. Mitigation of timing channels involves either neutralizing the channel completely or reducing its bandwidth to acceptable levels. The first step in covert channel analysis typically involves estimating the worst case bandwidth and the effect of various system parameters like noise and delay [35, 6, 16, 43, 34]. Once this is done, there are many ways in which channel bandwidth can be reduced, including the network pump [20, 21, 22], fuzzy time [18], timing jammers [16] and language transformations [5]. Reducing the bandwidth of covert channel does not imply that the covert channel threat is removed. Useful and important information like encryption keys can still be leaked out over low-bandwidth covert channel [31].

Because it is often not practical to neutralize covert timing channels completely, it might be preferable to detect their active exploitation rather than suffering the performance penalties associated with reducing their potential bandwidth [11]. The detection of network timing channel exploitation is known to be a difficult problem in general. Although specific methods [11, 7] have been proposed to handle various covert encodings, they do not work against every kind of timing channel. All these mechanisms rely on some notion of "regularity" to distinguish between regular network traffic and covert timing channel traffic. The exact regularity depends on the specific channel encoding to be detected and therefore, none of these methods work for every possible scheme.

Side-channel attacks against cryptosystems are somewhat similar to covert channels. Side-channel attacks exploit information leaked by an application's implementation of a crypto algorithm. By measuring the time it takes to perform different cryptographic operations and a knowledge of the implementation, it is sometimes possible to extract key bits [25]. It has been shown that side-channel timing attacks can be practical over a network [10]. Side-channel leakage can also occur in contexts outside of cryptographic algorithms themselves. Song et al. [41] describe a timing attack on the inter-keystroke timing of an interactive SSH connection. Their experiments indicate that one can gain 5.7 bits of information about an SSH password from the observed inter-keystroke timings over a network, assuming a password length of 8 characters. This corresponds to a 50x reduction in work factor for a bruce-force attack.

In fact, the most commonly studied examples of network timing channels in the recent literature are cryptosystem side-channel attacks. Here, the amount of information leaked per packet is very small but given sufficient data and large enough samples, it is possible to perform effective cryptanalysis [23].

Actual malicious attacks exploiting covert channels have not been commonly reported in the literature. Covert *storage* channels exploiting unused TCP/IP header fields have been used in the past by DDoS tools [13]. We are not aware of any public reports documenting the use of malicious covert network *timing* channels in the wild over the Internet, although it is at least plausible that they too have been exploited as part of real attacks.

Given the high variability in round trip times of network packets and their unreliable delivery mechanisms without any QoS guarantees, it is natural to ask whether covert timing channels are even practical on the Internet. Surprisingly, there has been relatively little research on the practical exploitation of covert network timing channels. Cabuk et al. [11] describe the design of a simple software-based network timing channel over IP. Because the timing channel is software based, the sender of the channel has complete control over the network subsystem. Their timing channel uses a binary symbol encoding where the presence or absence of a network packet in a timing interval signifies a bit of information.

The idea of perturbing the timing information of existing network traffic is not new. Addition of timing jitters to existing network packets has been studied previously for SSH stepping stone correlation [45] and for tracking VoIP calls [44]. VoIP tracking relies on encoding timing information in VoIP packets to encode a 24-bit watermark that can then be used to correlate two separate VoIP flows. This is made possible by exploiting the regularity of VoIP traffic and modifying the statistical properties of groups of packets to encode bits. Some timing attacks on anonymizing mix networks also rely on perturbing flows [36, 30].

3 Input Channels and Network Events

In the discussion that follows, we use the following terminology while talking about covert network timing channels. The *sender* of the channel is the subverted entity that is responsible for modulating the timing to encode information. It can be an application software, part of the operating system or a hardware device. The *receiver* in the channel can either be a network connection endpoint or a passive eavesdropper that extracts information from the channel by looking at network packet timings.

The sender in a covert network timing channel aims to modulate the timing of packets on the network to which the receiver has access. This may, for example, be the result of a software trojan that generates network packets at specific times corresponding to the information being sent [11]. Similarly, a router in the path of a network packet can change [44] the timing of the packets it receives before sending them to their destination. In both these examples, the sender of the timing channel has complete control over the network packets and can directly influence their timing on the network. Ideally, when the network delay is negligible, the receiver of the timing channel observes the same timings as those intended by the sender. Thus, the sender of the covert channel is a part of an already compromised output channel or device. Research in practical network timing channels typically considers such direct channel senders. This threat model, however, is overly conservative. It is possible to have usable and practical network timing channels that require only the compromise of system components that have traditionally been thought to lie comfortably within a host's security boundary: the input subsystems.

That the subversion of an input channel or device is a sufficient condition for a practical network timing channel to exist is a somewhat surprising claim. However, once we consider that many network events are directly caused by activity on input channels, it is easy to see how such covert channels might work. Also, because we are just interested in timing, we only need to modify the timing of existing input events. It is not necessary to generate any new traffic.

From the attacker's perspective, the goal of a covert channel is to leak secrets in violation of the host's security policy. Compromised input devices expose any secrets communicated over the input channel. For example, compromising a keyboard (used by the Keyboard JitterBug) allows the attacker to learn passphrases and other personal information that can then be leaked over the covert network timing channel.

In fact, compromising an output channel to leak secrets over a covert channel is not a very interesting scenario for the attacker. Once such a channel or device has been infiltrated by an attacker, leaking secrets from it is very easy. A compromised output subsystem has many options for communicating with an unauthorized receiver, often at much higher bandwidth than a covert channel could provide.

Input based channels do not fit well within the traditional model used in covert channel analysis. As we will see, their presence – as well as the fact that they can be exploited in practice – makes it necessary to include input devices in the Trusted Computing Base (TCB).

The coupling between input devices and the network is made possible by timing propagation often present in general purpose computing systems. Once these channels have been identified, they can be exploited with a JitterBug.

4 Networks and JitterBugs

Loosely coupled network timing channels and JitterBugs are a way of thinking about covert network channels in conventional computer architectures that emphasizes their potential for exploitation. As such, they also provide a model under which the threat of covert channels in conventional computer systems can be analyzed.

One of the characteristics of the software and router based network timing channel described in the previous section is that the sender and receiver of the channel are closely coupled together.

In *loosely coupled network timing channels*, the sender and receiver might be separated by multiple system layers, each belonging to a different level of abstraction. These channels are based on the observation that, just as data flow occurs in a general computing system, timing information also propagates from one system object to the other. By perturbing this timing information, it is possible to modulate a receiver many stages ahead in this flow. It is easier to see how this can be done by considering an example flow that is exploited by the Keyboard JitterBug.

Consider the case where the user is running an interactive network application. Each keypress triggers a sequence of events. The keyboard sends scan codes over the keyboard cable to the host's keyboard controller. This transmission is not instantaneous and depends on the state of the hardware, whether there's enough space in the keyboard controller buffer, etc. This in turn causes an interrupt to be generated to the operating system. Depending on the operations being performed, there might be a variable delay between when the value is received by the keyboard buffer and when it is read by the operating system. Once the interrupt handling routine has read the value from the keyboard controller, the operating system will typically perform some additional operations (e.g. scan-code → key-code translation) and put this value into a buffer to be read by the user-space network application, typically through a *read()* system call. Once the interactive network software gets the character, it might perform additional processing (e.g. encryption) before requesting the OS to send the character in a network packet. Similarly, additional delays will occur due to the network stack and hardware before the packet is sent out on the network. The timing of the network packet corresponds to the time when the key is pressed and the sum of all these additional delays.

In the above example, the flow of timing information (when the key is pressed) goes through several iterations of these added delays while the data moves through various system layers at different abstractions. Each of these layers adds noise to the timing information by imposing a non-deterministic delay due to their internal scheduling, buffering and processing mechanisms. Loosely coupled timing channels are based on the idea that the timing information can be influenced at any one of these several layers.

As long as the sender of the covert timing channel is positioned somewhere before or within any of these layers, it can modulate event timing to transmit data. The encoding applied by the sender is dependent on the properties of this channel that exists between itself and the receiver. The more the number of layers between the sender and receiver, the weaker is their coupling on the timing channel. A loosely coupled network timing channel is one where the source and the receiver of the timing channel are separated by many such delay inducing layers.

4.1 JitterBugs

JitterBugs are a class of mechanisms that can be used to covertly exploit a loosely coupled network timing channel. They have two defining properties. First, they have access to (and can recognize) sensitive information. Second, they have the ability to modulate event timing over a loosely-coupled network timing channel.

The covert transmission need not performed at the same time the sensitive information is captured. A JitterBug can collect and store sensitive information and replay it later over the loosely coupled network timing channel. A JitterBug is semi-passive in nature, i.e. it does not generate any new events. All modulation is done by piggybacking onto pre-existing events. This also makes a JitterBug much harder to detect in comparison to a more active covert timing channel source. Figure 1 shows the general architecture of a JitterBug.

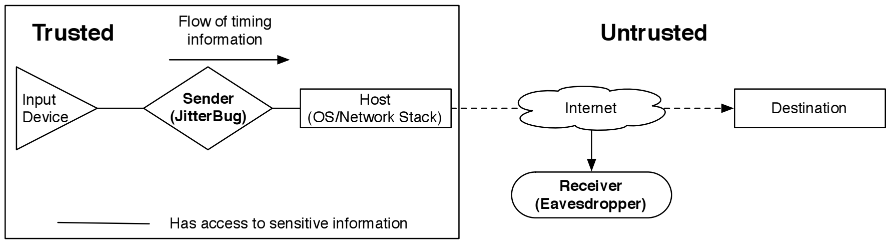


Figure 1: High-level overview of JitterBug

4.2 Example Channels

The keyboard is not the only channel susceptible to exploitation by a JitterBug. Other input peripherals can also provide a suitable environment for a covert network timing channel to exist. Various network computing applications allow users to remotely access hosts on the Internet as if they were being used locally. Some examples of such applications include NXClient, VNC (Virtual Network Computing) and Microsoft Remote Desktop. To minimize lag and keep the response time low, user input is typically transmitted over the network as soon as it is received on the sender's side. This timing channel can be exploited by placing a JitterBug between the communication path of the input device and the computer. Any digital input device – the mouse, digital microphone, web camera, etc. – is potentially exploitable in this way.

Many VoIP implementations support optimizations based on "silent intervals", periods of speech where nothing is being said. Network communication while using VoIP is typically regular. Packets with voice data are sent out at regular intervals over the network. If the silent interval feature is supported, then during periods of silence, packets are no longer sent to conserve bandwidth and system resources. By adding extraneous noise that influences the times at which these silent intervals are generated, a covert network timing channel can exist. In this case, a JitterBug can be placed in the audio interface or behind a digital microphone.

5 Keyboard JitterBug

In most interactive network applications (e.g. SSH, XServer, Telnet, etc.), each keypress corresponds to a packet (possibly encrypted) being sent out on the network. The timing of these packets is closely correlated with the times at which the keys were pressed. The Keyboard JitterBug adds small delays to keypresses that encode the data to be covertly exfiltrated. By observing the precise times packets arrive on the network, a remote receiver can recover this data. The Keyboard JitterBug does not generate any new network packets. It piggybacks its output atop existing network traffic by modulating timing.

The Keyboard JitterBug makes it possible to leak secrets over the network simply by compromising a keyboard input channel. It is, in effect, an advanced keylogger that solves the data exfiltration problem in a novel way.

5.1 Architecture

Our Keyboard JitterBug is implemented as a hardware interception device that sits between the keyboard and the computer. It is also possible to implement a JitterBug by modifying the keyboard firmware or the internal keyboard circuits, but the bump-in-the-wire implementation lends itself to easy installation on existing keyboards without the need for any major modification. Figure 2 shows the high-level architecture of the Keyboard JitterBug.

The Keyboard JitterBug adds timing information to keypresses in the form of small jitters that are unnoticeable to a human operator. If the user is typing in an interactive network application, then each keystroke will be sent in its own network packet. Ignoring the effects of buffering and network delays (the ideal case), the timing of the network packets will mirror closely the times at which the keystroke were received by the keyboard controller on the host. By observing these packet timings, an eavesdropper can reconstruct the original information that was encoded by the Keyboard JitterBug.

5.2 Symbol Encoding

The Keyboard JitterBug implements a covert timing channel by encoding information within inter-keystroke timings. By modifying the timing of keyboard events received by the keyboard controller, the inter-keystroke timings are manipulated such that they satisfy certain properties depending on the information it is trying to send.

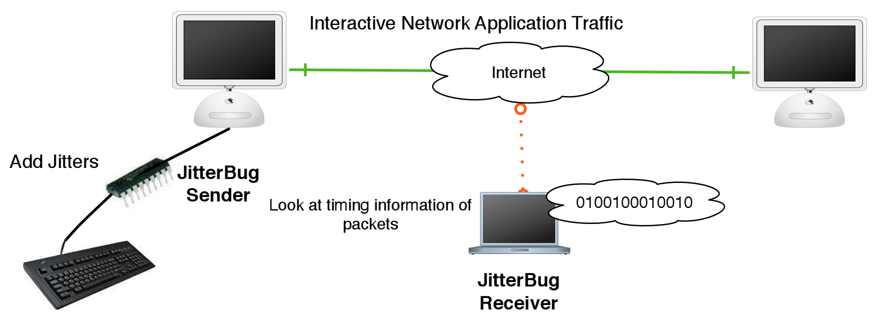

Interactive Network Application Traffic
Internet
Add Jitters
JitterBug Sender
Look at timing information of packets
0100100010010
JitterBug Receiver

Figure 2: Keyboard JitterBug architecture

The sender and the receiver do not require synchronized clocks but they do need clocks with sufficient accuracy. Our prototype Keyboard JitterBug uses its own crystal controlled clock to govern timing.

Below we describe a simple binary encoding scheme where each timing interval corresponding to adjacent keystrokes encodes a single bit of information.

To encode a binary sequence $\{b_i\}$ using the Keyboard JitterBug, we manipulate the sequence $\{t_i\}$ of times when the keystrokes are pressed by adding a delay denoted by τ_i to each element of this original sequence. The new sequence of events $\{t'_i\}$, where each $t'_i = t_i + \tau_i$, are the times at which the keystrokes are released by the Keyboard JitterBug to the keyboard controller. The resulting sequence encodes information in the differences $\delta_i = t'_i - t'_{i-1}$, such that:

$$\delta_i \bmod w = \begin{cases} 0 & \text{if } b_i = 0; \\ \lfloor w/2 \rfloor & \text{if } b_i = 1; \end{cases}$$

where w is a real-time parameter called the *timing window*.

Therefore, to encode a '0' the delay added is such that $\delta_i \bmod w$ is 0 and to encode a '1', the delay added is such that $\delta_i \bmod w$ is $\lfloor w/2 \rfloor$. In this symbol encoding scheme, within the timing window of length w, $\lfloor w/2 \rfloor$ is the antipode of 0.

Observe that each $\tau_i < w$. Hence, w defines the maximum delay added to each keystroke by the Keyboard JitterBug.

It is easy to understand the encoding algorithm with the help of a simple example. Assuming a window size w of 20 ms, to transmit the bit sequence $\{0, 1, 0, 1, 1\}$, the JitterBug would add delay such that the modified inter-keystroke timings (modulo 20) would be $\{0, 10, 0, 10, 10\}$. So if the (original) observed inter-keystroke timings were (in ms) $\{123, 145, 333, 813, 140\}$, the delay added would be such that the modified inter-keystroke timings are $\{140, 150, 340, 830, 150\}$. Hence, the JitterBug would use the delay sequence $\{17, 5, 7, 17, 10\}$ where each of these individual delays is less than 20 ms.

5.3 Symbol Decoding

For the Keyboard JitterBug network timing channel, the receiver is a passive eavesdropper that needs only the ability to measure the times at which each network packet arrives on the network. There are two ways a receiver might extract this timing information: *TCP Timestamps* and *sniffer timestamps*.

The *TCP Timestamp* option, described in RFC 1323 [19], allows each TCP packet to contain a 32-bit timestamp. This 32-bit TCP timestamp is a monotonically increasing counter and acts as a virtual-clock. In most modern operating systems, the TCP timestamp is directly derived from the system clock. The granularity of this clock depends on the operating system in use. Some commonly used values are 10 ms (some Linux distributions and FreeBSD), 500 ms (OpenBSD), and 100 ms (Microsoft Windows). As TCP timestamps correspond to the time at which the network packet was sent according to the source clock, they are unaffected by network jitter. The chief disadvantage of using TCP timestamps is their much coarser granularity on many operating systems, requiring the use of large timing windows for symbol encoding and decoding. Also, TCP timestamps are only used for a flow if both ends support the option and in addition, the initial SYN packet for the connection contained this option.

Sniffer timestamps, in contrast, represent the times at which packets are seen by a remote network sniffer. Due to network delays, these timestamps are offset from the actual time the packet was sent at the source. In addition, these timing offsets are affected by any network jitter present.

Based on the above discussion, it is clear that the choice of the particular timestamp to use depends on the exact network conditions, timing window size and the placement of the receiver on the network relative to the covert channel sender. However, we use sniffer timestamps exclusively for our experiments as they provide sufficient granularity for a much wider range of window sizes and operating systems. Also, since the Keyboard JitterBug has no control over the host or its OS, assuming only sniffer timestamps is a more conservative assumption for the attacker.

For decoding, the receiver on the timing channel records the sequence of times $\{\widehat{t_i}\}$ of network packets corresponding to each keystroke. Then the sequence of differences $\{\widehat{\delta_i} = \widehat{t_i} - \widehat{t_{i-1}}\}$, encodes the bits of information being transmitted. To allow the receiver to handle small variations in network transit times due to network jitter, the decoding algorithm allows some tolerance. The tolerance parameter ε is used by the decoder to handle these small fluctuations. The decoding algorithm is as follows:

$$\begin{array}{llll} \text{if} & -\varepsilon < \widehat{\delta_i} \leq \varepsilon & (\bmod w) & \text{then } b_i = 0; \\ \text{if} & w/2 - \varepsilon \leq \widehat{\delta_i} < w/2 + \varepsilon & (\bmod w) & \text{then } b_i = 1; \end{array}$$

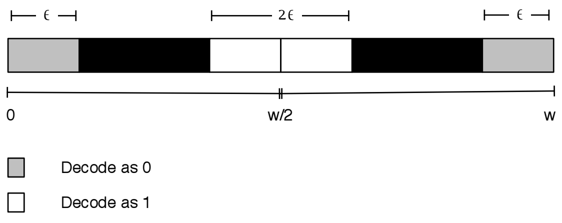


Figure 3: Timing Window for binary symbol decoding

The tolerance ε is an important parameter that decides the length of *guard bands* that compensate for the variability in the network and other delays. Figure 3 shows how the receiver decodes bits based on the inter-packet delays modulo the length of the timing window. The bands used for the decoding are calculated based on the value of ε. From the figure it is easy to see that maximum value of ε is $w/4$. Note that for a particular choice of w and ε, the proportion of timing window allocated for '1' and a '0' may not be equal.

For applications where the total added jitter is an important consideration, the tolerance ε can be used during symbol encoding to reduce the average jitter added at the cost of some channel performance.

The length of the timing window is an important parameter. We want it to be small so as to minimize the keyboard lag experienced by the user. At the same time, we want to make sure the guard bands are large enough to handle channel noise.

Because the receiver uses inter-packet delay and not absolute packet times, there is no need for synchronization between the source and receiver clocks. The clocks, however, need to run at the same rate.

The above scheme allows one bit of information to be transmitted per keypress. However, it is also possible to use a more efficient symbol alphabet with cardinality greater than two by subdividing the window further (instead of just two regions) corresponding to each possible symbol that can be transmitted. This choice however impacts the required granularity of the timing window. More specifically, for an encoding scheme with alphabet $\mathcal{A}$, cardinality k, and a tolerance of ε for each symbol, the timing window w needs to be atleast $2k\varepsilon$ units in length. We experimentally evaluate one such scheme in Section 6.3.6.

5.4 Framing and Error Correction

Our Keyboard JitterBug assumes that there will be bursts of contiguous keyboard activity in the interactive network application generating network packets, though these bursts themselves might be interrupted and infrequent. In our model, the only information sent over the covert timing channel is ASCII text corresponding to short user passphrases. Consequently, we do not perform any detailed analysis of the performance of the channel using different framing mechanisms. However, we tested the Keyboard JitterBug using two very simple framing schemes.

One approach is based on *bit-stuffing* [28], which uses a special sequence of bits known as the Frame Sync Sequence (FSS) that acts as frame delimiter. This sequence is prevented from occurring in the actual data being transmitted by "stuffing" additional bits when it is encountered in the data stream. Conversely, these extra bits are "destuffed" by the decoder at the receiver to recover the original bits of information. The advantage of using bit-stuffing is that it does not require any change to the underlying low-level symbol encoding scheme. For example, the symbol alphabet can still be binary.

An alternative framing mechanism adds a third symbol to the low-level encoding scheme. This special symbol in the underlying transmission alphabet acts as a frame delimiter. Note that if the length of the timing window is kept constant, this reduces the maximum possible length of the guard bands used for decoding the information symbols (0 and 1) compared to a purely binary scheme. So this might lead to lower channel performance if network noise is present. It is also useful to give more tolerance to the frame delimiter symbol encoding as framing errors cause the whole frame to be discarded at the receiver. Thus, delimiter corruption causes a much higher commensurate effect on the overall bit error rate than the corruption of a single bit. This issue is discussed further in Section 6.3.

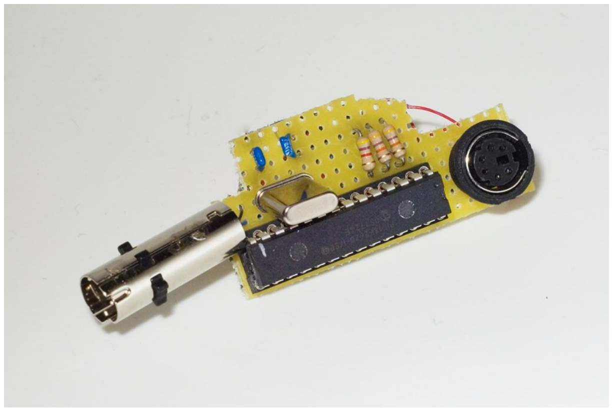


Figure 4: Prototype Keyboard JitterBug

Error correction might also be required if the timing channel suffers from a lot of noise. However, in the simple case in which a short encryption key or password is being leaked, forward error correction is provided inherently by repeating the transmission each time it completes.

5.5 Prototype PIC implementation

We implemented a prototype Keyboard JitterBug on the Microchip PIC18F series of Peripheral Interface Controllers (PICs). The PIC18F series is a family of 12-bit core flash programmable microcontrollers. Our source code is a combination of C and PIC assembly and the final machine code uses less than 5KB program memory. The implementation works for keyboards that use the IBM PS/2 protocol. It should be easy to port the code to other kinds of keyboards, e.g. USB. The bump-in-the-wire implementation acts as a relay device for PS/2 signals. It derives its power from the PS/2 voltage lines and hence no additional power source is required. When enabled, it adds jitters to delay the time at which the keyboard controller receives notification of the keypress. It also supports programmable triggers (as described in Section 5.6) that help identify typed sensitive information to leak over the covert channel. Figure 4 shows our PIC-based prototype implementation. A truly surreptitious Keyboard JitterBug would have to be small enough to conceal within a cable or connector. Since the computational requirements are quite modest here, miniaturization could be readily accomplished through the use of smaller components or with customized ASIC hardware.

5.6 Attack scenarios

We consider a real and somewhat famous example from recent news reports that motivated our design. In gathering evidence in the 2000 bookmaking case [3] against Nicodemo “Little Nicky” Scarfo, the FBI surreptitiously installed some sort of keylogger device in the suspect’s computer to gain access to his PGP passphrases. Installing the device apparently required physical access to the suspect’s office, a high-risk and expensive operation. Once installed, the device recorded keypresses under certain conditions. This introduced a new problem: retrieval of the captured information. A conventional keylogger must either compromise the host software (to allow remote access and offloading of captured data) or require physical access to recover the device itself. Neither option is entirely satisfactory from the FBI’s perspective. Compromise of the host software creates an ongoing risk of discovery or data loss (if the host software is updated or replaced), and physical recovery requires additional (risky) physical access. The Keyboard JitterBug adds a third option: leaking the targeted information atop normal network traffic through the timing channel, obviating the need for subsequent retrieval or compromise of the host.[1]

As the Keyboard JitterBug lies in the communication path between the keyboard and the computer system, it has access to the keystrokes typed in by the user. The covert network timing channel is relatively low-bandwidth and thus the JitterBug needs the capability to recognize and store the specific information of interest with high confidence. JitterBug’s programmable triggers do just that by acting as recognizers of sensitive information (like passphrases) and storing this information for sending out later over the covert network timing channel. Programmable triggers allow a Keyboard JitterBug to wait for particular strings to be typed. When such a condition is detected, it stores whatever string is typed next into its internal EEPROM for subsequent transmission.

For example, a Keyboard JitterBug might be programmed with the user name of the target as the trigger, on the assumption that the following keystrokes are likely to include a password. It might also be programmed to detect certain typing patterns that tend to indicate that the user is initiating an SSH connection (e.g. “ssh username@host”). By storing whatever is subsequently typed by the user, the Keyboard JitterBug effectively gets hold of the user’s SSH password. The covert channel transmits the password back to the attacker without the need to retrieve the bug; the password can even be sent atop the victim’s own encrypted SSH connection.

In this sense, Keyboard JitterBug can be seen as a next step in the evolution of keyloggers. The possibil-

[1]Because the Scarfo case never went to trial, the technology used by the FBI to capture the keystrokes was never publicly disclosed – it may have been a JitterBug, although it was more likely a conventional keylogger. The PGP passphrase of interest turned out to be based on Mr. Scarfo’s father’s US Prison ID number.

ity of such devices raises obvious privacy and security concerns.

The Keyboard JitterBug implementation can also serve as the basis for more advanced worms and viruses. Many newer keyboards are software programmable. Some of these keyboards even allow their internal firmware to be upgraded by software. A malicious virus program might rewrite the firmware with a JitterBug(ged) version and delete itself, effectively avoiding any form of detection by an antivirus program.

Finally, perhaps the most serious (and also the most sophisticated to mount) application of the JitterBug is as part of a *Supply Chain Attack*. Rather than targeting a specific system, the attacker subverts the keyboard supply and manufacturing process to install such a device in many keyboards from one or more suppliers, in the hope that a compromised device will eventually be acquired by a target of interest. Such an attack seems most plausible in the context of government espionage or information warfare, but could also be mounted by an industrial or individual attacker who manages to compromise a keyboard vendor's code base.

5.7 Non-interactive network applications

Although the Keyboard JitterBug's primary application is for leaking secrets or other information over interactive network applications, it can also be used in a restricted setting with very low bandwidth for less interactive network applications. Much network activity has a causal relationship with specific keyboard events. This is true for many commonly used network programs such as web browsers, instant messengers and email clients.

For IM programs, pressing return after a line of text causes the message to be sent over a network. In addition, many IM protocols also send a notification to the other end as soon as the user starts typing another line. By detecting and manipulating keystroke timings when such events happen, the Keyboard JitterBug can leak information. Similarly, typing a URL into a web-browser typically requires the user to press "return" before the browser fetches it. The Keyboard JitterBug can manipulate this timing to affect the time at which the URL is fetched over the network. The relevant "return" when the jitter should be added can be detected by using a programmable trigger (e.g. Ctrl-L → URL→<return> for Mozilla Firefox). E-mail clients also sometimes use keyboard shortcuts which cause specific network events (e.g. sending an e-mail) to occur. By adding jitter to the appropriate keypresses, the timing of these network events can be manipulated (and observed).

For the above applications, the coupling between keyboard events and network activity make them susceptible to attacks using the Keyboard JitterBug. The bandwidth of leakage, however, is significantly lower. One advantage they have over SSH from the perspective of the attacker is that many of these applications tend not to use encryption. This reduces the number of insertion errors (Section 6.2) by making it easier for the covert channel receiver to distinguish between normal network packets and those whose timing was manipulated by the Keyboard JitterBug.

6 Keyboard JitterBug: Evaluation

In this section, our focus is on evaluating the efficacy of the timing channel under a variety of practical conditions.

6.1 Factors affecting performance

Because the JitterBug is so far removed from its receiver, many factors affect its performance.

- **Buffering:** Keyboard buffering affects the delay between when the key is received by the keyboard controller and when it is available to the application that is trying to send the keystroke over the network. Similarly, network buffering affects the delay between when the request for sending the packet is received by the OS network stack and when it is actually transmitted over the network. If the variance of buffering delay (keyboard + network) is high, then the number of symbol errors increase, reducing the effective bitrate of the channel.

- **OS Scheduling:** For a loosely-coupled covert timing channel, the noise added by OS scheduling depends on a variety of factors including the time quantum used, the scheduling algorithm, system load, etc. Fortunately, keyboard and network handling in most modern operating systems is given high priority and hence, the noise added to the channel from scheduling effects is usually quite insignificant.

- **Nagle's algorithm:** Described in RFC 896 [37], Nagle's algorithm is used to handle the *small-packet problem* that is caused by the increase in packet header overhead when interactive network applications are used as each keystroke is sent in its own network packet. The algorithm is an adaptive way of deciding when to buffer data before sending it out in a single network packet based on the network conditions (latency and bandwidth). If Nagle's algorithm is enabled it can cause two problems with the timing channel. Firstly, it creates a varying network buffering delay that adds noise to the timing information. Secondly, it can lead to multiple keystrokes being sent out in a single packet. Hence, the timing information for all but the first keystroke might be lost leading to missing symbols in the timing channel. Fortunately, Nagle's algorithm is usually disabled by default (using the TCP_NODELAY option) for better responsiveness in interactive network applications including most commonly used SSH client implementations (e.g. OpenSSH). This means that each keystroke generates it own network packet that is sent out as soon as possible (assuming no network congestion).

- **Network Jitter:** This is the most important factor for a network timing channel. Network Jitter, i.e. variability in round trip times (RTT), adds noise to the timing information and affects the accuracy of symbol decoding at the receiver. The placement of the receiver also affects the "observed" network jitter and thus changes the observed channel accuracy. Encoding a symbol in the timing of two adjacent packets has a mitigating effect on the channel accuracy as each change in network delay causes a maximum of one error to occur.

6.2 Sources of Error

The Keyboard JitterBug timing channel can suffer from three kinds of transmission errors: insertions, deletions and inversions.

Insertions occur when receiver cannot distinguish between network packets corresponding to the Keyboard JitterBug and those corresponding to other network traffic. This will happen when any form of encryption is being used. Depending on the protocol layer at which encryption is being applied, the frequency of insertion errors will be different. The worst case is when link encryption is being used. In this case, it would be very hard to separate covert channel packets with that of normal network traffic, causing insertion errors to happen all the time. Fortunately, the use of link layer encryption along the whole path of a network packet on the Internet is quite rare, so this restriction is not that much of an issue. Encryption at the network or transport layers (e.g. IPSec, TLS) would also cause significant insertion errors to occur, especially if one of the network applications of interest use them for communication. Application layer encryption can cause insertion errors but they are pretty rare as the visible packet format and size (e.g. SSH) makes it possible (in most cases) to distinguish packets of interest from normal network traffic. Finally, if no encryption is being used (e.g. telnet), then no insertion errors occur.

Deletion errors are of two kinds. As the Keyboard JitterBug only has access to keystrokes and no other system information, it is not possible to distinguish between when the user is typing inside a network application of interest or in other applications running on the system. The situation can be ameliorated somewhat by using heuristics to determine when the user is typing in a network application (e.g. by detecting shell commands being typed when previously the user opened up a new ssh connection) and add jitters only then. In cases where this is not possible, multiple chunks of bits might be lost. The second kind of deletion errors occur when network buffering causes multiple keystrokes to be sent in the same packet. These deletion errors occur less frequently and typically cause very few symbols to be lost. They can always be detected when no encryption is being used (e.g. telnet). For the more general case, an appropriate framing scheme would be required.

The main application of the Keyboard JitterBug channel is to leak passwords, typed cryptographic keys, and other such secrets. As these secrets are relatively short, they can be transmitted repeatedly to increase the chances that they will be received correctly. Both insertion and deletion errors are, by their nature, bursty. The redundancy through repetition provides inherent forward error correction (FEC) to handle them.

Finally, symbol corruption errors are caused by delays that might occur on the sender's side or in the network while the packet is in transit (due to network jitter). These errors cause a different symbol to be received than what was originally sent. For the binary symbol encoding scheme, the errors take the form of bit inversions. Symbol corruption errors can be handled by using suitable error correction coding schemes.

As insertion and deletion errors are very specific to the application and environment under which the Keyboard JitterBug is deployed, we do not focus on them in our experimental evaluation.

6.3 Experimental Results

We performed various experiments to test the Keyboard JitterBug under a variety of sender configurations, network and receiver conditions. The experiments were performed with our bump-in-the-wire implementation of the Keyboard JitterBug on a PIC microcontoller.

As our covert channel relies on manipulating the timing of keypresses to piggyback information, the keyboard needs to be in use for the channel to work and be tested. Instead of manually typing at the keyboard for each experiment, we built a keyboard replayer for our controlled experiments. A special mode in the Keyboard JitterBug allows it to store all keyboard traffic into the EEPROM for later replay. Then the covert timing channel can be turned on and the replay information is used to simulate a real user typing at the keyboard preserving the original user's keystroke timing information. This way we can test different Keyboard JitterBug parameters under the same set of conditions. Note that the Keyboard JitterBug is still placed as a relay device between the keyboard and the computer. The available memory of the PIC device limits the maximum length of the replay. When the end of a replay is reached, the JitterBug starts the replay from the beginning. This does not materially affect our experiments, since we are concerned only with the inter-character timing, not the actual text.

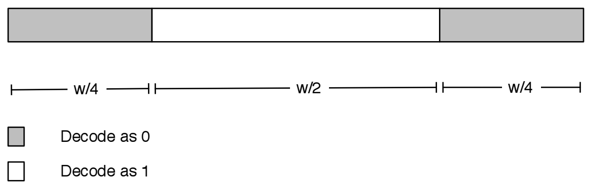


Figure 5: Timing Window ($\varepsilon = w/4$) used for binary symbol decoding in experiments

For all experiments where a pure binary symbol encoding is being used, the user-defined tolerance parameter $\varepsilon = w/4$. Figure 5 shows the decoding timing window used with the bands for '0' and '1'.

The source machines used for the experiments were connected to the LAN network at the Dept. of Computer and Information Science, University of Pennsylvania, Philadelphia. The source machines ran Linux 2.4.20 (unless otherwise noted). All network connections were made via a 100Mbps switch. As we are interested in finding how well the Keyboard JitterBug performs under a range of different network conditions, we used the PlanetLab network [12] to test our covert network timing channel using various geographically displaced nodes. Interactive SSH terminal sessions were initiated between the source and destination nodes. All measurements of the timing information for the covert channel were performed at the destination host using *tcpdump*. Using the time of arrival of network packets at the destination host gives us a worst case estimate of the channel performance. In practice, the covert channel receiver can be placed anywhere in the path of the network packets. The channel is configured to send an ASCII encoded string.

The standard measure of the performance of channel under the presence of noise is the bit error rate (BER) [40]. For channels with bit slips[2], due to the possibility of bit loss, this metric cannot directly be used. For the Keyboard JitterBug, as network buffering can cause more than one keystroke to be sent in each packet, there is potential for missing bits leading to synchronization errors. Therefore, while measuring raw channel performance (without framing or error correction), the traditional definition of bit error rate based on the Hamming Distance metric cannot be used. Instead, we use *Levenshtein Distance*, also called the *edit distance* to get the raw bit error rate. Here, an error constitutes inversion or deletion of bits. The edit distance is a measure of similarity of two strings and is equal to the number of insertions, deletions, and substitutions needed to convert the source string (bits received) into the target string (bits sent).

[2]In general, the lack of synchronization might occur for various other reasons, such as the lack of buffer space, variation in clock rate, etc.

While measuring channel performance with framing, the bit error rate is calculated using the Hamming Distance metric for correctly received frames. For frames discarded because of framing errors, all the data bits (of the frame) are assumed to have been in error. Because of framing, the receiver can detect and recover from bit deletions and synchronize itself with the covert channel data stream. For evaluating the performance of the channel with framing, three parameters are calculated: Net BER (E_C), Average Correct Frame BER (E_{CF}) and Frame Discard Rate (E_{DF}). Net BER measures the total fraction of bits that are lost or corrupted due to bit errors within a frame or framing errors caused due to corruption of the Frame Sync Sequence or delimiter. Framing errors cause whole frame(s) to be discarded leading to the loss of all bits they contain. These bit errors (equal to the frame size) are included in the calculation for Net BER. Average Correct Frame BER is the average BER only for the frames that were successfully decoded (without framing errors). Therefore, bits lost due to framing errors are not accounted for in calculating the Average Correct Frame BER. The suitable error correction coding scheme to use would depend on this measure. Frame Discard Rate is a measure of the frequency with which frames get dropped or lost due to framing errors. It is easy to see that:

$$E_C = E_{CF} + E_{DF} - E_{CF}E_{DF}$$

6.3.1 Window Size and RTT

Table 1 summarizes the measured raw BER of the covert network timing channel for six different nodes on the PlanetLab network using different window sizes. These nodes were chosen based on their wide ranging geographical distances from the source host and different network round-trip times.

The raw BER is the channel performance without the use of any error correction coding or framing. As the calculation of the raw BER metric uses the edit distance metric, the error rates also consider bit deletions and insertions in addition to inversions. The notion of acceptable raw channel performance would depend on a variety of factors including the framing mechanism used, the application, and the capability of error correction codes.

Node	RTT	Hops	1s	500ms	100ms	20ms	15ms	10ms	5ms	2ms
ColumbiaU (NYC, NY)	6 ms	14	0	0	0	.007	.007	.010	.044	.089
UKansas (Lawrence, KS)	42 ms	14	0	0	0	.005	.007	.008	.067	.143
UUtah (Salt Lake City, UT)	73 ms	23	0	0	0	.005	.005	.005	.039	.092
UCSD (San Diego, CA)	84 ms	19	0	0	0	.010	.011	.011	.044	.102
ETHZ (ETH, Zurich)	112 ms	17	0	0	0	.005	.006	.009	.049	.092
NUS (Singapore)	236ms	18	0	0	0	.045	.047	.048	.228	.240

Table 1: **Measured Raw Bit Error Rate for different window sizes and network nodes (Levenshtein Distance Metric)**

Many error correction codes exist for channels where both substitutions and deletions are possible and that use the Levenshtein distance metric as the error rate metric [29]. Marker Codes [15, 39] and Watermark Codes [14] are some examples of such error correction schemes. As our primary application for the channel is very low-bandwidth, we consider a measured raw bit error rate of less than 10% to be acceptable. We discuss channel performance using the Hamming distance metric in Section 6.3.5 when we discuss experiments with the use of some simple framing schemes.

For a fixed window size, the round-trip times and the channel performance do not exhibit any clear trend. Intuitively, this lack of a trend is to be expected. The channel encoding relies on the packet inter-arrival times for encoding the information. Thus, it is the network jitter and not the end-to-end latency that affects performance of the channel.

Acceptable performance is achievable even if the receiver is at a large distance from the source of the timing channel. The node in Singapore, with a RTT of 236 ms, is a case in point. For a window size of 20 ms, the raw channel error rate is around 4.5%, which is quite usable for many low-bandwidth applications of the JitterBug.

The maximum lag introduced by the Keyboard JitterBug for each keypress is equal to the window size w. Consequently, the choice of this parameter is dependent upon how large the value can be made while still keeping the Keyboard JitterBug undetectable by the user. Although we can get a perfect channel for all the nodes tested with a window size of 1 second, this value is effectively unusable because the user will detect the presence of the Keyboard JitterBug. It is widely believed that 0.1 seconds is about the limit for the response time for a user to feel that the system is reacting instantaneously [32]. Therefore in practice, the window size will have to be smaller than that. Our own experience with the Keyboard JitterBug shows that 20 ms is a perfectly acceptable window size and this amount of added lag for each keystroke is effectively unnoticeable by the user.

The window size also affects the size of the guard bands that help absorb some network jitter. The jitter has two components: the frequency of change and the magnitude of change. For a window size of w the implementation can handle a maximum jitter of $w/4$ per packet pair.

Load	20ms	15ms	10ms	5ms
SSH	.010	.011	.011	.044
Telnet	0	.006	.01	.01

Table 2: **Measured Raw Bit Error Rate for SSH and Telnet (Levenshtein Distance Metric)**

From Table 1, it is clear that, as expected, smaller window sizes lead to higher error rates. The increase in the error rate, however, is not very drastic over the ranges we tested. The channel remain usable even if window sizes as low as 2 ms are used. For a window size of 20 ms or more, channel performance is consistently high on all the nodes tested. Our observations are supported by previous studies of round-trip delays on the Internet. It has been shown that on average, round-trip delays on the Internet tend to cluster around within a jitter window of 10 ms for significant periods of time [4]. Thus, this choice of window size is likely to work under a wide gamut of network conditions. When the exact conditions are known, it is possible to optimize the Keyboard JitterBug further by choosing smaller window sizes.

6.3.2 Network application

We measured the raw BER for four different windows sizes for a covert timing channel to a PlanetLab node in University of California, San Diego. The node is 19 hops away with an average Round-Trip Time (RTT) of 84.3 ms. Table 2 shows the measured raw BER for SSH and Telnet. The channel performance is not affected by the choice of the interactive network terminal application. The advantage of Telnet, of course, is its lack of encryption, which makes it easy to detect deletion errors caused by multiple characters being sent in the same network packet.

OS	20ms	15ms	10ms	5ms
Linux 2.4.20	.010	.011	.011	.044
Linux 2.6.10	.010	.010	.010	.013
Windows XP(SP2)	.001	.001	.001	.007
FreeBSD 5.4	.017	.033	.044	.058
OpenBSD 3.8	.022	.043	.05	.075

Table 3: **Measured Raw Bit Error Rate for different window sizes and operating systems (Levenshtein Distance Metric)**

Load	20ms	15ms	10ms	5ms
Idle	.010	.011	.011	.044
Heavy Load	.010	.016	.016	.05

Table 4: **Measured Raw Bit Error Rate for different windows sizes and system loads (Levenshtein Distance Metric)**

6.3.3 Operating System

To confirm that the performance of the channel is not significantly affected by the operating system through which the Keyboard JitterBug is working, we performed experiments to measure the performance of the implementation on several popular operating systems.[3] We again performed the experiments on the PlanetLab node at San Diego, California for four different window sizes. Table 3 summarizes the measured raw BER of the covert timing channel for different operating systems. The raw BER remains quite similar for all the operating systems tested without any major fluctuations. The small difference in the results arises from two factors: variations in network conditions and different OS implementations of keyboard processing. Both these factors affect the amount of noise present in the timing channel when it reaches the receiver.

6.3.4 System Load

Keyboard and network events in general-purpose operating systems are typically given high processing priority. Moreover, their implementation is usually interrupt-driven for better responsiveness and performance. So, we do not expect the normal variation in system loads to have any major influence on the performance of the timing channel. To confirm this, we used the *stress* [1] tool to generate high system loads[4] at the source machine and then measured the performance of the timing channel at the receiver. As before, the receiver of the timing channel is located at the PlanetLab Node in San Diego, CA. The source of the timing channel is a Pentium 4 2.4 GHz Desktop System with 1GB of system memory running Linux 2.4.20.

Table 4 shows the measured raw BER for normal system load vs. heavy system load. The results show that the behavior of the channel remains quite similar without any drastic drops in the channel performance.

[3]We did not perform experiments with Mac OS X because of the absence of a PS/2 keyboard port on the Mac hardware.

[4]The command-line used was: *stress –cpu 8 –io 4 –vm 2 –vm-bytes 256M*

Node	$\mathbf{E_T}$	$\mathbf{E_{CF}}$	$\mathbf{E_{DF}}$
ColumbiaU (NYC, NY)	.142	0	.142
UKansas (Lawrence, KS)	.152	0	.152
UUtah (Salt Lake City, UT)	.093	0	.093
UCSD (San Diego, CA)	.184	0	.184
ETHZ (ETH, Zurich)	.112	0	.112
NUS (Singapore)	.384	.014	.375

Table 5: **Measured Bit Error Rate(s) with Framing (Bit-Stuffing)** (E_T = Net BER, E_{CF}: Average Correct Frame BER, E_{DF}: Frame Discard Rate)

Node	$\mathbf{E_T}$	$\mathbf{E_{CF}}$	$\mathbf{E_{DF}}$
ColumbiaU (NYC, NY)	.121	.002	.12
UKansas (Lawrence, KS)	.104	0	.104
UUtah (Salt Lake City, UT)	.137	.001	.136
UCSD (San Diego, CA)	.202	.001	.2
ETHZ (ETH, Zurich)	.088	0	.088
NUS (Singapore)	.39	.005	.386

Table 6: **Measured Bit Error Rate(s) with Framing (Ternary Encoding)** (E_T = Net BER, E_{CF}: Average Correct Frame BER, E_{DF}: Frame Discard Rate)

6.3.5 Framing

Many applications of the Keyboard JitterBug would require the use of framing for transmission of data on the timing channel. We tested the JitterBug with two very simple framing schemes: one based on bit stuffing and the other using a low-level special frame delimiter symbol. Our goal is to evaluate the performance of the channel using the Hamming distance metric rather than describe an optimal framing scheme for the timing channel.

The timing window used for the experiments is 20 ms and the frame size is 16 bits. The bit-stuffing frame sync sequence (FSS) used is 8 bits in length. The results are summarized in Table 5 and Table 6. As described in Section 6.3, three parameters are calculated for each run: the Net BER, Average Correct Frame BER and the Frame Discard Rate. The receiver discards any frame that is not the correct size or has a corrupted frame delimiter.

It is clear from the results that the bulk of the network errors are the result of discarded frames. Many of these are synchronization errors caused by deletion of bits from a frame due to network buffering. There are

Node	$\mathbf{E_T}$	$\mathbf{E_{CF}}$	$\mathbf{E_{DF}}$
ColumbiaU (NYC, NY)	.150	.011	.140
UKansas (Lawrence, KS)	.174	.030	.148
UUtah (Salt Lake City, UT)	.170	.012	.16
UCSD (San Diego, CA)	.173	.021	.156
ETHZ (ETH, Zurich)	.153	.007	.147
NUS (Singapore)	.34	.057	.299

Table 7: **Measured Bit Error Rate(s) with high bit-rate encoding (4bits/symbol + frame delimiter)** (E_T = Net BER, E_{CF}: Average Correct Frame BER, E_{DF}: Frame Discard Rate)

many possible ways the framing scheme could be optimized to reduce the frequency of framing errors. Using smaller frame sizes can reduce the affect of discarded frames on the overall BER. One could also use a much more optimistic decoder so that partial frames are not discarded completely but parts of their contents are recovered. This would most likely need to be combined with an error correction coding scheme for the data within the frame. Coding schemes based on either the Hamming distance metric (to handle substitutions) or Levenshtein distance metric [42] (to handle deletions as well) could be used. Another approach would be to modify the framing scheme to reduce the chance of frame corruption. For example, using two frame delimiters at the start of every frame instead of one. This way if only one of the delimiters gets deleted or corrupted, the frame can still be decoded correctly.

6.3.6 Encoding Scheme

Our results for smaller window sizes indicate that for many environments in which the Keyboard JitterBug might be deployed, one could use a more efficient symbol encoding scheme by packing more than one bit of information with each transmitted symbol. To confirm this hypothesis, we implemented a 16 symbol (four bits/symbol) encoding scheme with an additional symbol acting as the frame delimiter. The results of our experiments are summarized in Table 7. The frame size used is 16 bits (four symbols). The Average Correct Frame BER stays at above acceptable levels for all the nodes tested. The results show that it is possible to optimize the framing and encoding schemes to increase the bandwidth of the channel and at the same time maintain acceptable channel performance.

6.4 Summary of the results

Our experimental results indicate that a conservative choice of the window size as 20 ms is small enough to be undetectable by a normal user and at the same time gives good channel performance under a variety of system loads, operating systems and network conditions. One can also increase the bandwidth of the channel by choosing a more aggressive encoding scheme as our results for the high bit rate encoding show. However, our primary goal was to design an encoding scheme that is robust and general enough to work under any unknown environment without affecting user perception. The binary encoding scheme with a timing window of 20 ms serves that purpose quite well.

6.5 Detection

The detection of covert network timing channels is a separate research problem of its own and as such, quite difficult. Thus we do not focus on the detectability aspects of the channel in this paper. However, we briefly analyze some of the issues.

It has been suggested in previous studies that covert network timing channels can be detected by looking at the inter-arrival times of network packets [11, 7]. These detection algorithms rely on the notion of *regularity*, a channel-specific property that can be used to distinguish normal traffic from certain kinds of covert channel traffic. None of these techniques work for detecting the presence of *any* covert timing channel. The Keyboard JitterBug is a low-bandwidth timing channel and has a different form of regularity. Hence, these techniques are unlikely to be able to detect the exploitation of our timing channel.

However, it might be possible to detect Keyboard JitterBug activity by directly observing the inter-arrival times of network packets. The inter-arrival times tend to cluster around multiples of the window size or half the window size. This is because the symbol encoding scheme relies on using an inter-arrival time of 0 (modulo w) for sending a '0' and $w/2$ (modulo w) for sending a '1'. We collected an SSH trace without the use of a Keyboard JitterBug. We then modified the trace by adding simulated jitter so that packet timings corresponded to the case when a Keyboard JitterBug is being used. Because we do not model the effect of noise added by network jitter, this gives us a worst case analysis of the detectability of our channel.

Figure 6 shows the inter-arrival times for 550 packets in the original trace for a range between 0.2s and 0.3s. In Figure 7, we show the same trace except now with simulated jitter that would be added by a Keyboard JitterBug. Notice the banding around multiples of 10 ms, which corresponds to a window size of 20 ms. Thus, a simple plot of the inter-arrival times reveals that that a covert timing channel is being exploited.

To evade such a simple detection scheme, an approach based on rotating the timing window used for symbol encoding is described below. Note, however, that we do

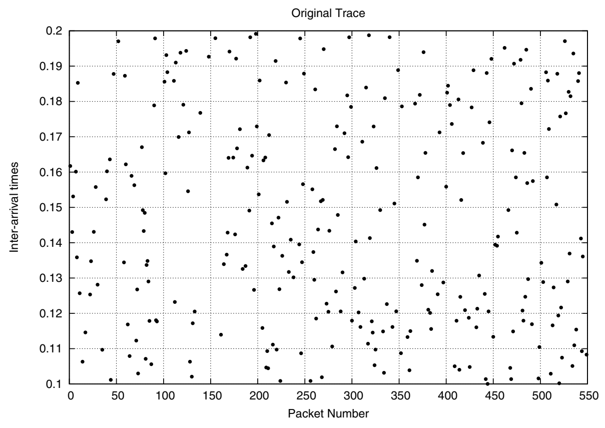


Figure 6: Original SSH Trace

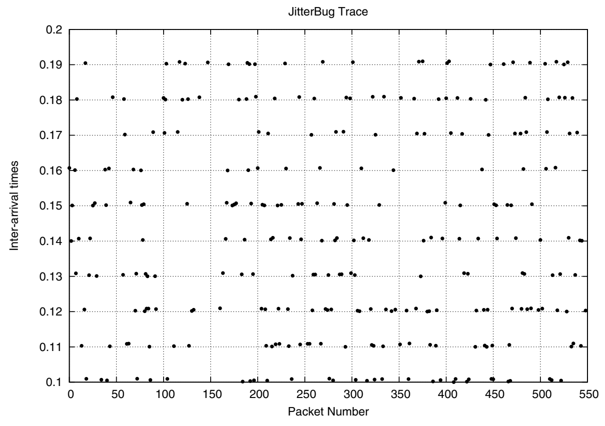


Figure 7: JitterBug applied to the original SSH Trace (stationary time windows)

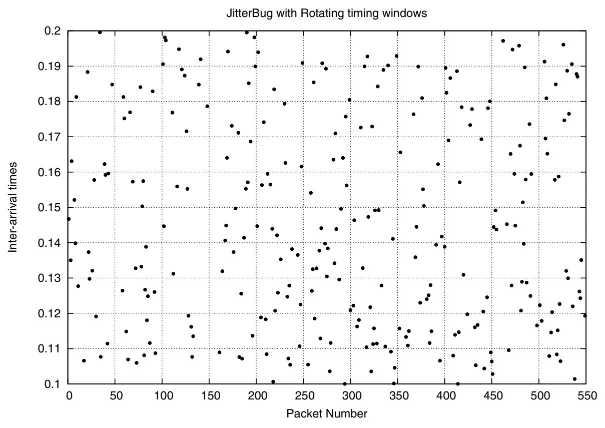


Figure 8: JitterBug applied to the original SSH Trace (rotating time windows)

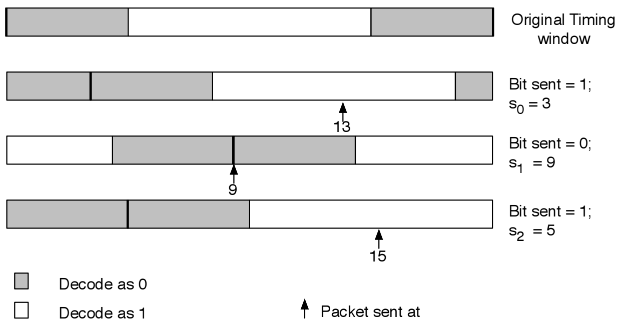


Figure 9: Rotating timing windows: The symbol encoding window is rotated for sending each bit

not claim that the use of the following technique makes our channel undetectable using any other technique. It is simply a countermeasure against the most direct way of detecting our covert timing channel. The timing channel might still be susceptible to other forms of analysis that detect its presence in network traffic.

The method works as follows. As before, let us denote by $\{b_i\}$ the binary sequence to be transmitted using jitters, and by $\{t_i\}$ the sequence of the times when the keys are pressed. Assume there exists $\{s_i\}$, a pseudo-random sequence of integers that range from 0 to $w-1$, where w is, as before, the length of the timing window. The sequence $\{s_i\}$ is assumed to be known by the sender and the receiver but not by anyone else, and works as a shared secret. Rather than encoding bits by adding delays so that the inter-arrival distances cluster around 0 and its antipode, the source adds jitter such that they cluster around the sequence $\{s_i\}$ and its associated antipodal sequence.

More precisely, in order to transmit the bit b_i, the JitterBug adds a delay such that:

$$(\delta_i - s_i) \bmod w = \begin{cases} 0 & \text{if } b_i = 0; \\ \lfloor w/2 \rfloor & \text{if } b_i = 1; \end{cases}$$

where $\delta_i = t_i^{'} - t_{i-1}^{'}$, as before are the difference in times when adjacent keystrokes are sent to the keyboard controller by the Keyboard JitterBug.

Consider an example where Bob wants to send 3-bits of information $\{1,0,1\}$ to Eve using JitterBug. Assume that the window size is 20 ms, and that they agreed on the sequence $\{s_0, s_1, s_2\} = \{3, 9, 5\}$. Figure 9 illustrates how the timing window is rotated at each step before deciding on the amount of jitter to add.

Figure 8 shows the inter-arrival times for the same SSH trace with packet timing adjusted for JitterBug but this time using rotating windows during symbol encoding instead of the original static scheme. The sequence $\{s_i\}$ is chosen to be a pseudo-random sequence of integers between 0 and 19. The inter-arrival times are no longer clustered now and there are no new noticeable patterns compared to the original SSH trace.

The intuition behind this approach is that the resulting sequence $\{\widehat{\delta_i}\}$ on the receiver's side looks as arbitrary as $\{s_i\}$. The choice of $\{s_i\}$ is obviously important and should be sufficiently random . Note that when $\{s_i = 0 \; ; \forall \; i\}$, this reduces to the original case with a stationary time window.

7 Conclusions and Future Work

Compromising an input channel is useful not only for learning secrets, but, as we have seen, is also often sufficient for leaking them over the network. We introduced *loosely-coupled network timing channels* and *JitterBugs*, through which covert network timing channels can be exploited to leak sensitive information in general-purpose computing systems. We described the *Keyboard JitterBug*, our implementation of such a network timing channel. The Keyboard JitterBug is a keylogger that does not require physical retrieval to exfiltrate its captured data. It can leak previously captured sensitive information such as user passphrases over interactive network applications by adding small and unnoticeable delays to user keypresses. It is even possible to use the Keyboard JitterBug, at low-bandwidth with other, non-interactive, network applications, such as web browsers and instant messaging systems.

Our experiments suggest that the distance over the network between the receiver and the JitterBug doesn't matter very much. The timing window size w is the basic parameter of the symbol encoding scheme. Its choice is dictated by the expected amount of jitter in the network and by the maximum delay that can be tolerated. A conservative choice of the window size as 20 ms is small enough to be unnoticeable to a human user and at the same time gives good channel performance over a wide range of network conditions and operating systems tested. This makes a Keyboard JitterBug very robust and less susceptible to major changes in the environment in which it is installed. We also described experimental results with some simple framing schemes and more aggressive encoding mechanisms. Our results show that the symbol encoding and framing could be further optimized for better performance in certain environments. Finally, we showed simple techniques for defeating the most direct ways of detecting our attacks.

The most obvious extension to this work is the development of better framing and encoding schemes with higher bandwidth, by making less conservative assumptions that take advantage of specific channel properties. In this paper, however, we deliberately avoided optimizing for any particular channel, operating system, or networked application, instead identifying parameters that give satisfactory performance and that remain highly robust under varied conditions.

All covert timing channels represent an arms race between those who exploit such channels and those who want to detect their use. This necessitates the use of countermeasures by a covert channel to elude detection by network wardens. We suggested only very simple countermeasures in this paper. Our initial results with rotating encoding timing windows indicate that the use of cryptographic techniques to hide the use of encoded jitter channels may be a promising approach. We plan to explore this direction in the future.

Acknowledgments

This research was supported in part by grants from NSF Cybertrust (CNS-05-24047) and NSF SGER (CNS-05-04159). Jutta Degener suggested the name "JitterBug". The idea of using a PIC chip to add jitters emerged from discussions with John Ioannidis. We thank Madhukar Anand, Sandy Clark, Eric Cronin, Chris Marget and Micah Sherr for the many helpful discussions during the course of this research. We are grateful for the facilities of PlanetLab to perform our experiments. Finally, we thank the anonymous reviewers and David Wagner for many helpful suggestions and comments.

References

[1] The *stress* project. http://weather.ou.edu/ apw/projects/stress/.

[2] Trusted computer system evaluation. Tech. Rep. DOD 5200.28-STD, U.S. Department of Defense, 1985.

[3] United States v. Scarfo, Criminal No. 00-404 (D.N.J.), 2001.

[4] ACHARYA, A., AND SALZ, J. A Study of Internet Round-Trip Delay. Tech. Rep. CS-TR-3736, University of Maryland, 1996.

[5] AGAT, J. Transforming out timing leaks. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2000), ACM Press, pp. 40–53.

[6] ANANTHARAM, V., AND VERDU, S. Bits Through Queues. In *IEEE Transactions On Information Theory* (1996), vol. 42.

[7] BERK, V., GIANI, A., AND CYBENKO, G. Detection of Covert Channel Encoding in Network Packet Delays. Tech. rep., Darthmouth College, 2005.

[8] BROIDO, A., HYUN, Y., AND KC CLAFFY. Spectroscopy of traceroute delays. In *Passive and active measurement workshop* (2005).

[9] BROIDO, A., KING, R., NEMETH, E., AND KC CLAFFY. Radon spectroscopy of inter-packet delay. In *IEEE high-speed networking workshop* (2003).

[10] BRUMLEY, D., AND BONEH, D. Remote Timing Attacks are Practical. In *Proceedings of the 12th USENIX Security Symposium* (August 2003).

[11] CABUK, S., BRODLEY, C. E., AND SHIELDS, C. IP covert timing channels: design and detection. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security* (New York, NY, USA, 2004), ACM Press, pp. 178–187.

[12] CHUN, B., CULLER, D., ROSCOE, T., BAVIER, A., PETERSON, L., WAWRZONIAK, M., AND BOWMAN, M. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev. 33*, 3 (2003), 3–12.

[13] DAEMON9. Project Loki. *Phrack Magazine 7*, 49 (August 1996).

[14] DAVEY, M. C., AND MACKAY, D. J. Reliable communication over channels with insertions, deletions, and substitutions. *IEEE Transactions on Information Theory 47* (2001).

[15] F. F. SELLERS, J. Bit loss and gain correction code. In *IEEE Transactions on Information Theory* (1962), vol. 8, pp. 35–38.

[16] GILES, J., AND HAJEK, B. An Information-Theoretic and Game-Theoretic Study of Timing Channels. In *IEEE Transactions on Information Theory* (2002), vol. 48.

[17] HELOUET, L., JARD, C., AND ZEITOUN, M. Covert channels detection in protocols using scenarios. In *Proceedings of SPV '2003, Workshop on Security Protocols Verification* (2003). Satellite of CONCUR'03. Available at `http://www.loria.fr/~rusi/spv.pdf`.

[18] HU, W.-M. Reducing Timing Channels with Fuzzy Time. In *IEEE Symposium on Security and Privacy* (1991).

[19] JACOBSON, V., BRADEN, R., AND BORMAN, D. RFC 1323 - TCP Extensions for High Performance.

[20] KANG, M. H., AND MOSKOWITZ, I. S. A Data Pump for Communication. Tech. rep., Naval Research Laboratory, 1995.

[21] KANG, M. H., MOSKOWITZ, I. S., AND LEE, D. C. A Network Version of the Pump. In *IEEE Symposium on Security and Privacy* (1995).

[22] KANG, M. H., MOSKOWITZ, I. S., MONTROSE, B. E., AND PARSONESE, J. J. A Case Study Of Two NRL Pump Prototypes. In *ACSAC '96: Proceedings of the 12th Annual Computer Security Applications Conference* (Washington, DC, USA, 1996), IEEE Computer Society, p. 32.

[23] KELSEY, J., SCHNEIER, B., WAGNER, D., AND HALL, C. Side Channel Cryptanalysis of Product Ciphers. In *ESORICS '98* (1998).

[24] KEMMERER, R. A. A Practical Approach to Identifying Storage and Timing Channels: Twenty Years Later. In *ACSAC '02: Proceedings of the 18th Annual Computer Security Applications Conference* (Washington, DC, USA, 2002), IEEE Computer Society, p. 109.

[25] KOCHER, P. C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO* (1996), pp. 104–113.

[26] KOHNO, T., BROIDO, A., AND KC CLAFFY. Remote Physical Device Fingerprinting. In *IEEE Symposium on Security and Privacy* (2005).

[27] LAMPSON, B. W. A Note on the Confinement Problem. In *Communications of the ACM* (1973), vol. 16.

[28] LEE, P. *Combined error-correcting/modulation recording codes*. PhD thesis, Univesity of California, San Diego, 1988.

[29] LEVENSHTEIN, V. I. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet Physics Doklady* (1966), vol. 10, pp. 707–710.

[30] LEVINE, B., REITER, M., WANG, C., AND WRIGHT, M. Timing Attacks in Low-Latency Mix Systems. In *Proceedings of Financial Cryptography: 8th International Conference (FC 2004): LNCS-3110* (2004).

[31] MILLEN, J. 20 years of covert channel modeling and analysis. In *IEEE Symposium on Security and Privacy* (1999).

[32] MILLER, R. B. Response time in man-computer conversational transactions. In *AFIPS Fall Joint Computer Conference* (1968), vol. 33.

[33] MOSKOWITZ, I. S., AND KANG, M. H. Covert Channels – Here to Stay ? In *COMPASS* (1994).

[34] MOSKOWITZ, I. S., AND MILLER, A. R. The Influence of Delay Upon an Idealized Channel's Bandwidth. In *SP '92: Proceedings of the 1992 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 1992), IEEE Computer Society, p. 62.

[35] MOSKOWITZ, I. S., AND MILLER, A. R. Simple timing channels. In *IEEE Symposium on Security and Privacy* (1994).

[36] MURDOCH, S., AND DANEZIS, G. Low-cost traffic analysis of tor. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy* (2005).

[37] NAGLE, J. RFC 896 - Congestion Control in IP/TCP Internetworks.

[38] PROCTOR, N. E., AND NEUMANN, P. G. Architectural Implications of Covert Channels. In *15th National Computer Security Conference* (1992).

[39] RATZER, E. A., AND MACKAY, D. J. C. Codes for channels with insertions, deletions and substitutions. In *Proceedings of 2nd International Symposium on Turbo Codes and Related Topics, Brest, France, 2000* (2000), pp. 149–156.

[40] SHANNON, C. E. A mathematical theory of communication. *Bell System Technical Journal* (1948), 379–423 and 623–656.

[41] SONG, D. X., WAGNER, D., AND TIAN, X. Timing analysis of keystrokes and timing attacks on ssh. In *USENIX Security Symposium* (2001).

[42] TANAKA, E., AND KASAI, T. Synchronization and substitution error-correcting codes for the Levenshtein metric. In *IEEE Transactions on Information Theory* (March 1976), vol. 22, pp. 156–162.

[43] VENKATRAMAN, B. R., AND NEWMAN-WOLFE, R. Capacity Estimation and Auditability of Network Covert Channels. In *IEEE Symposium on Security and Privacy* (1995).

[44] WANG, X., CHEN, S., AND JAJODIA, S. Tracking anonymous peer-to-peer VoIP calls on the internet. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security* (New York, NY, USA, 2005), ACM Press, pp. 81–91.

[45] WANG, X., AND REEVES, D. Robust Correlation of Encrypted Attack Traffic Through Stepping Stones by Manipulation of Interpacket Delays. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003)* (2003).

[46] WRAY, J. C. An Analysis of Covert Timing Channels. In *Proceedings of the IEEE Symposium on Research in Security and Privacy, Oakland, California* (1991).

Lessons from the Sony CD DRM Episode

J. Alex Halderman and Edward W. Felten
Center for Information Technology Policy
Department of Computer Science
Princeton University

Abstract

In the fall of 2005, problems discovered in two Sony-BMG compact disc copy protection systems, XCP and MediaMax, triggered a public uproar that ultimately led to class-action litigation and the recall of millions of discs. We present an in-depth analysis of these technologies, including their design, implementation, and deployment. The systems are surprisingly complex and suffer from a diverse array of flaws that weaken their content protection and expose users to serious security and privacy risks. Their complexity, and their failure, makes them an interesting case study of digital rights management that carries valuable lessons for content companies, DRM vendors, policymakers, end users, and the security community.

1 Introduction

This paper is a case study of the design, implementation, and deployment of anti-copying technologies. We present a detailed technical analysis of the security and privacy implications of two systems, XCP and MediaMax, which were developed by separate companies (First4Internet and SunnComm, respectively) and shipped on millions of music compact discs by Sony-BMG, the world's second largest record company. We consider the design choices the companies faced, examine the choices they made, and weigh the consequences of those choices. The lessons that emerge are valuable not only for compact disc copy protection, but for copy protection systems in general.

The security and privacy implications of Sony-BMG's CD digital rights management (DRM) technologies first reached the public eye on October 31, 2005, in a blog post by Mark Russinovich [21]. While testing a rootkit detector he had co-written, Russinovich was surprised to find an apparent rootkit (software designed to hide an intruder's presence [13]) on one of his systems. Investigating, he found that the rootkit was part of a CD DRM system called XCP that had been installed when he inserted a Sony-BMG music CD into his computer's CD drive.

News of Russinovich's discovery circulated rapidly on the Internet, and further revelations soon followed, from us,[1] from Russinovich, and from others. It was discovered that the XCP rootkit makes users' systems more vulnerable to attacks, that both CD DRM schemes install risky software components without obtaining informed consent from users, that both systems covertly transmit usage information back to the vendor or the music label, and that none of the protected discs include tools for uninstalling the software. (For these reasons, both XCP and MediaMax seem to meet the consensus definition of spyware.) These and other findings outraged many users.

As the story was picked up by the popular press and public pressure built, Sony-BMG agreed to recall XCP discs from stores and to issue uninstallers for both XCP and MediaMax, but we discovered that both uninstallers created serious security holes on users' systems. Class action lawsuits were filed soon after, and government investigations were launched, as Sony-BMG worked to repair relations with its customers.

While Sony-BMG and its DRM vendors were at the center of this incident, its implications go beyond Sony-BMG and beyond compact discs. Viewed in context, it is a case study in the deployment of DRM into a mature market for recorded media. Many of the lessons of CD DRM apply to other DRM markets as well.

Several themes emerge from this case study: similarities between DRM and malicious software such as spyware, the temptation of DRM vendors to adopt malware tactics, the tendency of DRM to erode privacy, the strategic use of access control to control markets, the failure of ad hoc designs, and the force of differing incentives in shaping behavior and causing conflict.

Outline The remainder of the paper is structured as follows. Section 2 discusses the business incentives of record labels and DRM vendors, which drive their technology decisions. Section 3 gives a high-level technical summary of the systems' design. Sections 4–9 each cover one aspect of the design in more detail, discussing the design choices made in XCP and MediaMax and considering alternative designs. We discuss weaknesses in the copy protection schemes themselves, as well as vulnerabilities they introduce in users' systems. We cover installation issues in Section 4, recognition of protected discs in Section 5, player software in Section 6, deactivation attacks in Section 7, uninstallation issues in Section 8, and compatibility and upgrading issues in Section 9. Section 10 explores the outrage users expressed in response to the DRM problems. Section 11 concludes and draws lessons for other systems.

2 Goals and Incentives

The goals of a CD DRM system are purely economic: the system is designed to protect and enable the business models of the record label and the DRM vendor. Accordingly, any discussion of goals and incentives must begin and end by talking about business models. The record label and the DRM vendor are separate actors whose interests are not always aligned. Incentive gaps between the label and the DRM vendor can be important in explaining the design and deployment of CD DRM systems.

2.1 Record Label Goals

We first examine the record label's goals. Though the label would like to keep the music from the CD from being made available on peer-to-peer (P2P) file sharing networks, this goal is not feasible [4]. If even one user can rip an unprotected copy of the music and put it on a P2P network, it will be available to the whole world. In practice, every commercially valuable song appears on P2P networks immediately upon release, if not sooner. No CD DRM system can hope to stop this. Real systems do not appear designed to stop P2P sharing, but seem aimed at other goals.[2]

The record label's goal must therefore be to retard disc-to-disc copying and other local copying and use of the music. Stopping local copying might increase sales of the music—if Alice cannot copy a CD to give to Bob, Bob might buy the CD himself.

Control over local uses can translate into more revenue for the record label. For example, if the label can control Alice's ability to download music from a CD into her iPod, the label might be able to charge Alice an extra fee for iPod downloads. Charging for iPod downloads creates new revenue, but it also reduces the value to users of the original CD and therefore reduces revenue from CD sales. Whether the new revenue will outweigh the loss of CD revenue is a complex economic question that depends on detailed assumptions about users' preferences; generally, increasing the label's control over uses of the music will tend to increase the label's profit.

Whether the label would find it more profitable to control a use, as opposed to granting it for free to CD purchasers, is a separate question from whether copyright law gives the label the right to file lawsuits relating to that use. Using DRM to enforce copyright law exactly as written is almost certainly not the record label's profit-maximizing strategy.

Besides controlling use of the music, CD DRM can make money for the record label because it puts software onto users' computers, and the label can monetize this installed platform. For example, each CD DRM album includes a special application for listening to the protected music. This application can show advertisements or create other promotional value for the label; or the platform can gather information about the user's activities, which can be exploited for some business purpose. If taken too far, these become spyware tactics; but they may be pursued more moderately, even over user objections, if the label believes the benefits outweigh the costs.

2.2 DRM Vendor Goals

The CD DRM vendor's primary goal is to create value for the record label in order to maximize the price the label will pay for the DRM technology. In this respect, the vendor's and label's incentives are aligned.

However, the vendor's incentives diverge from the label's in at least two ways. First, the vendor has a higher risk tolerance than the label, because the label is a large, established business with a valuable brand name, while the vendor (at least in the cases at issue here) is a start-up company with few assets and not much brand equity. Start-ups face many risks already and are therefore less averse to taking on one more risk. The record label, on the other hand, has much more capital and brand equity to lose if something goes horribly wrong. Accordingly, we can expect the vendor to be much more willing to accept security risks than the label.

The second incentive difference is that the vendor can monetize the installed platform in ways the record label cannot. For example, once the vendor's DRM software is installed on a user's system, the software can control use of other labels' CDs, so a larger installed base makes the vendor's technology more attractive to other labels. This extra incentive to build the installed base will make the vendor more aggressive about pushing the software onto users' computers than the label would be.

In short, incentive differences make the vendor more likely than the label to (a) cut corners and accept security risks, and (b) push DRM software onto more users' computers. If the label had perfect knowledge about the vendor's technology, this incentive gap would not be an issue—the label would simply insist that the vendor protect the label's interests. But if, as seems likely in practice, the label has imperfect knowledge of the technology, then the vendor will sometimes act against the label's interests. (For a discussion of differing incentives in another content protection context, see [9].)

2.3 DRM and Market Power

DRM affects more than just the relationships among the label, the vendor, and the user. It also impacts the label's and vendor's positions in their industries, in ways that will shape the companies' DRM strategies.

For example, DRM vendors are in a kind of standards war—a company that controls DRM standards has power to shape the online music business. DRM vendors fight this battle by spreading their platforms widely. Record labels want to play DRM vendors off against each other and prevent any one vendor from achieving dominance.

Major record companies such as Sony-BMG are parts of larger, diversified companies, and can be expected to help bolster the competitive position of their corporate siblings. For example, parts of Sony sell portable music players in competition with Apple, so Sony-BMG has an incentive to take steps to weaken Apple's market power.

Having examined the goals and motivations of the record labels and DRM vendors, we now turn to a description of the technologies they deployed.

3 CD DRM Systems

CD DRM systems must meet difficult requirements. Copy protected discs must be reasonably compliant with the CD Digital Audio standard so that they can play in ordinary CD players. They must be unreadable by almost all computer programs in order to prevent copying, yet the DRM vendor's own software must be able to read them in order to give the user some access to the music.

Most CD DRM systems use both passive and active anti-copying measures. Passive measures change the disc's contents in the hope of confusing most computer drives and software, without confusing most audio CD players. Active measures, in contrast, rely on software on the computer that actively intervenes to block access to the music by programs other than the DRM vendor's own software.

Active protection software must be installed on the computer somehow. XCP and MediaMax use Windows autorun, which (when enabled) automatically loads and runs software from a disc when the disc is inserted into the computer's drive. Autorun lets the DRM vendor's software run or install immediately.

Once the DRM software is installed, every time a new CD is inserted the software runs a recognition algorithm to determine whether the disc is associated with the DRM scheme. If it is, the active protection software will interfere with accesses to the disc, except those originating from the vendor's own music player application. This proprietary player application, which is shipped on the disc, gives the user limited access to the music.

As we will discuss further, all parts of this design are subject to attack by a user who wants to copy the music illegally or who wants to make uses allowed by copyright law but blocked by the DRM. The user can defeat the passive protection, stop the DRM software from installing itself, trick the recognition algorithm, defeat the active protection software's blocking, capture the music from the DRM vendor's player, or uninstall the protection software.

The complexity of today's CD DRM software offers many avenues of attack. On the whole, today's systems are no more resistant to attack than were simpler early CD DRM systems [10, 11]. When there are fundamental limits to security, extra complexity does not mean extra security.

Discs Studied Sony deployed XCP on 52 titles (representing more than 4.7 million CDs) [1]. We examined three of them in detail: Acceptance, *Phantoms* (2005); Susie Suh, *Susie Suh* (2005); and Switchfoot, *Nothing is Sound* (2005). MediaMax was deployed on 37 Sony titles (over 20 million CDs) as well as dozens of titles from other labels [1]. We studied three albums that used MediaMax version 3—Velvet Revolver, *Contraband* (BMG, 2004); Dave Matthews Band, *Stand Up* (Sony, 2005); and Anthony Hamilton, *Comin' from Where I'm From* (Arista/Sony 2005)—and three albums that used MediaMax version 5—Peter Cetera, *You Just Gotta Love Christmas* (Viastar, 2004); Babyface, *Grown and Sexy* (Arista/Sony, 2005); and My Morning Jacket, *Z* (ATO/Sony, 2005). Unless otherwise noted, statements about MediaMax apply to both version 3 and version 5.

4 Installation

Active protection measures cannot begin to operate until the DRM software is installed on the user's system. In this section we consider attacks that either prevent installation of the DRM software, or capture music files from the disc in the interval after the disc has been inserted but before the DRM software is installed on the computer.

4.1 Autorun

Both XCP and MediaMax rely on the autorun feature of Windows. Whenever removable media, such as a floppy disc or CD, is inserted into a Windows PC (and autorun is enabled), Windows looks on the disc for a file called `autorun.inf` and executes commands contained in it. Autorun is commonly used to pop up a splash screen or simple menu (for example) to offer to install software found on the disc. However, the autorun mechanism will run any program that the disc specifies.

Other popular operating systems, including MacOS X and Linux, do not have an autorun feature, so this mechanism does not work on those systems. XCP ships only Windows code and so has no effect on other operating systems. MediaMax ships with both Windows and MacOS code, but only the Windows code can autorun. The MacOS code relies on the user to double-click an installer, which few users will do. For this reason, we will not discuss the MacOS version of MediaMax further.

Current versions of Windows ship with autorun enabled by default, but the user can choose to disable it. Many security experts advise users to disable autorun to protect against disc-borne malware. If autorun is disabled, the XCP or MediaMax active protection software will not load or run. Even if autorun is enabled, the user can block autorun for a particular disc by holding down the Shift key while inserting the disc [11]. This will prevent the active protection software from running.

Even without disabling autorun, a user can prevent the active protection software from loading by covering up the portion of the disc on which it is stored. Both XCP and MediaMax discs contain two sessions, with the first session containing the music files and the second session containing DRM content, including the active protection software and the autorun command file. The first session begins at the center of the disc and extends outward; the second session is near the outer edge of the disc. By covering the outer edge of the disc, the user can prevent the drive from reading the second session's files, effectively converting the disc back to an ordinary single-session audio CD. The edge of the disc can be covered with non-transparent material such as masking tape, or by writing over it with a felt-tip marker [19]. Exactly how much of the disc to cover can be determined by iteratively covering more and more until the disc's behavior changes, or by visually inspecting the disc to look for a difference in appearance of the disc's surface which is often visible at the boundary between the two sessions.

4.2 Temporary Protection

Even if the copy protection software is allowed to autorun, there is a period of time, between when a protected disc is inserted and when the active protection software is installed, when the music is vulnerable to copying. It would be possible to have the discs immediately and automatically install the active protection software, minimizing this window of vulnerability, but legal and ethical requirements should preclude this option. Installing software without first obtaining the user's consent appears to be illegal in the U.S. under the Computer Fraud and Abuse Act (CFAA) as well as various state anti-spyware laws [2, 3].

Software vendors conventionally obtain user consent to the installation of their software by displaying an End User License Agreement (EULA) and asking the user to accept it. Only after the user agrees to the EULA is the software installed. The EULA informs the user, in theory at least, of the general scope and purpose of the software being installed, and the user has the option to withhold consent by declining the EULA, in which case no software is installed. As we will see below, the DRM vendors do not always follow this procedure.

If the discs didn't use any other protection measures, the music would be vulnerable to copying while the installer waited for the user to accept or reject the EULA. Users could just ignore the installer's EULA window and switch tasks to a CD ripping or copying application. Both XCP and MediaMax employ temporary protection mechanisms to protect the music during this time.

4.2.1 XCP Temporary Protection

The first time an XCP-protected disc is inserted into a Windows machine, the Windows autorun feature launches the XCP installer, the file `go.exe` located in the `contents` folder on the CD. The installer displays a license agreement and prompts the user to accept or decline it. If the user accepts the agreement, the installer installs the XCP active protection software onto the machine; if the user declines, the installer exits after ejecting the CD, preventing other applications from ripping or copying it.

While the EULA is being displayed, the XCP installer continuously monitors the list of processes running on the system. It compares the image name of each process to a blacklist of nearly 200 ripping and copying applications hard coded into the `go.exe` program. If one or more blacklisted applications are running, the installer replaces the EULA display with a warning indicating that the applications need to be closed in order for the installation to continue. It also initiates a 30-second countdown timer; if any of the applications are still running when the countdown reaches zero, the installer ejects the CD and quits.[3]

This technique might prevent some unsophisticated users from copying the disc while the installer is running, but it can be bypassed with a number of widely known techniques. For instance, users might kill the installer process (using the Windows Task Manager) before it can eject the CD, or they might use a ripping or copying application that locks the CD tray, preventing the installer from ejecting the disc.

The greatest limitation of the XCP temporary protection system is the blacklist. Users might find ripping or copying applications that are not on the list, or they might use a blacklisted application but rename its executable file to prevent the installer from recognizing it. Since there is no mechanism for updating the blacklist on existing CDs, they will gradually become easier to rip and copy as new applications not on the blacklist come into widespread use. Application developers may also adapt their software to the blacklisting technique by randomizing their process image names or taking other measures to avoid detection.[4]

4.2.2 MediaMax Temporary Protection

MediaMax employs a different—and highly controversial—temporary protection measure. It defends the music while the installer is running by installing, and at least temporarily activating, the active protection software *before* displaying the EULA. The software is installed without obtaining consent, and it remains installed (and in some cases, permanently active) even if the user explicitly denies consent by declining the license agreement.

MediaMax discs install the active protection driver by copying a file called `sbcphid.sys` to the Windows drivers directory, configuring it as a service in the registry, and launching it. Initially, the driver's startup type is set to "Manual," so it will not re-launch the next time the computer boots; however, it remains running until the computer is shut down, and it remains installed permanently [11]. Albums that use MediaMax version 5 additionally install components of the MediaMax player software before displaying a license agreement. These files are not removed if the EULA is declined.

Even more troublingly, under some common circumstances—for example, if the user inserts a MediaMax version 5 CD and declines the EULA and later inserts a MediaMax CD again—the MediaMax installer will permanently activate the active protection software (by setting its startup type to "Auto," which causes it to be launched every time the computer boots). This behavior is related to a mechanism in the installer apparently intended to upgrade the active protection software if an older version is already installed.

We can think of two possible explanations for this behavior. Perhaps the vendor, SunnComm, did not test these scenarios to determine what their software did, and so did not realize that they were activating the software without consent. Or perhaps they did know what would happen in these cases and deliberately chose these behaviors. Either possibility is troubling, indicating either a deficient design and testing procedure or a deliberate decision to install software after the user denied permission to do so.

Even if poor testing is the explanation for *activating* the software without consent, it is clear that SunnComm deliberately chose to *install* the MediaMax software on the user's system even if the user did not consent. These decisions are difficult to reconcile with the ethical and legal requirements on software companies. But they are easy to reconcile with the vendor's platform building strategy, which rewards the vendor for placing its software on as many computers as possible.

Even if no software is *installed* without consent, the temporary *activation* of DRM software, by both XCP and MediaMax, before the user consents to anything raises troubling ethical questions. It is hard to argue that the user has consented to loading running software merely by the act of inserting the disc. Most users do not expect the insertion of a music CD to load software, and although many (but not all) of the affected discs did contain a statement about protection software being on the discs, the statements generally were confusingly worded, were written in tiny print, and did not say explicitly that software would install or run immediately upon insertion of the disc. Some in the record industry argue that the industry's desire to block potential infringement justifies the short-term execution of the temporary protection software on every user's computer. We think this issue deserves more ethical and legal debate.

4.3 Passive Protection

Another way to prevent copying before active protection software is installed is to use passive protection measures. Passive protection exploits subtle differences between the way computers read CDs and the way ordinary CD players do. By changing the layout of data on the CD, it is sometimes possible to confuse computers without affecting ordinary players. In practice, the distinction between computers and CD players is imprecise. Older generations of CD copy protection, which relied entirely on passive protection, proved easy to copy in some computers and impossible to play on some CD players [10]. Furthermore, computer hardware and software has tended to get better at reading the passive protected CDs over time as it has become more robust to all manner of damaged or poorly formatted discs. For these reasons, more recent CD DRM schemes rely mainly on active protection.

XCP uses a mild variety of passive protection as an added layer of security against ripping and copying. This form of passive protection exploits a quirk in the way Windows handles multisession CDs. When CD burners came to market in the early 1990s, the multisession CD format was introduced to allow data to be appended to partially recorded discs. (This was especially desirable at a time when recordable CD media cost tens of dollars per disc.) Each time data is added to the disc, it is written as an independent series of tracks called a session. Multisession compatible CD drives see all the sessions, but ordinary CD players, which generally do not support the multisession format, recognize only the first session.

Some commercial discs use a variant of the multisession format to combine CD audio and computer accessible files on a single CD. These discs adhere to the Blue Book or "stamped multisession" format. According to the Blue Book specification, stamped multisession discs must contain two sessions: a first session with 1–99 CD audio tracks, and a second session with one data track. The Windows CD audio driver contains special support for Blue Book discs. It presents the CD to player and ripper applications as if it were a normal audio CD. Windows treats other multisession discs as data-only CDs.

XCP discs deviate from the Blue Book format by adding a second data track in the second session. This causes Windows to treat the disc as a regular multisession data CD, so the primary data track is mounted as a file system, but the audio tracks are invisible to player and ripper applications that use the Windows audio CD driver. This includes Windows Media Player, iTunes, and most other widely used CD applications. We developed a procedure for creating discs with this passive protection using only standard CD burning hardware and software.

This variety of passive protection provides only limited resistance to ripping and copying. There are a number of well-known methods for defeating it:

- *Advanced ripping and copying applications* avoid the Windows CD audio driver altogether and issue commands directly to the drive. This allows programs such as Nero and Exact Audio Copy to recognize and read all the audio tracks.
- *Non-Windows platforms,* including MacOS and Linux, read multisession CDs more robustly and do not suffer from the limitation that causes ripping problems on Windows.
- The *felt-tip marker trick*, described above, can also defeat this kind of passive protection. When the second session is obscured by the marker, CD drives see only the first session and treat the disc as a regular audio CD, which can be ripped or copied.

5 Disc Recognition

The active protection mechanisms employed by XCP and MediaMax regulate access to raw CD audio, blocking access to the audio tracks on albums protected with a particular scheme while allowing access to all other titles.

To accomplish this, the schemes install a background process that interposes itself between applications and the original CD driver. In MediaMax, this process is a kernel-mode driver called `sbcphid.sys`. XCP uses a pair of filter drivers called `crater.sys` and `cor.sys` that attach to the CD-ROM and IDE devices [21]. In both schemes, the active protection drivers examine each disc that is inserted into the computer to see whether access to it should be restricted. If the disc is recognized as copy protected, the drivers monitor for attempts to read the audio tracks, as would occur during a playback, rip, or disc copy operation, and corrupt the audio returned by the drive to degrade the listening experience. MediaMax introduces a large amount of random jitter, making the disc sound like it has been badly scratched or damaged; XCP replaces the audio with random noise.

Each scheme's active protection software interferes with attempts to rip or copy any disc that is protected by the same scheme, not merely the disc from which the software was installed. This requires some mechanism for identifying discs that are to be protected. In this section we discuss the security requirements for such a recognition system, and describe the design and limitations of the actual recognition mechanism employed by the MediaMax scheme.

5.1 Recognition Requirements

Any disc recognition system detects some distinctive feature of discs protected by a particular copy protection scheme. Ideally such a feature would satisfy four requirements: it would *uniquely* identify protected discs without accidentally triggering the copy protection on other titles; it would be *detectable* quickly after reading a limited amount of audio from the disc; it would be *indelible* enough that an attacker could not remove it without significantly degrading the quality of the audio; and it would be *unforgeable*, so that it could not be applied to an unprotected album without the cooperation of the protection vendor, even if the adversary had access to protected discs.

This last requirement stems from the DRM vendor's platform building strategy, which tries to put the DRM software on to as many computers as possible and to have the software control access to all marked discs. If the vendor's identifying mark is forgeable, then a record label could mark its discs without the vendor's permission, thereby taking advantage of the vendor's platform without paying.[5]

5.2 MediaMax Disc Recognition

To find out how well the disc recognition mechanisms employed by CD DRM systems meet the ideal requirements, we examined the recognition system built into MediaMax. This system drew our attention because MediaMax's creators have touted their advanced disc identification capabilities, including the ability to identify individual tracks within a compilation as protected [16]. XCP appears to use a less sophisticated disc recognition system based on a marker stored in the data track of protected discs; we did not include it in this study.

We determined how MediaMax identifies protected albums by tracing the commands sent to the CD drive with and without the active protection software running. These experiments took place on a Windows XP VMWare virtual machine running on top of a Fedora Linux host system, which we modified by patching the kernel IDE-SCSI driver to log all CD device activity.

With this setup we observed that the MediaMax software executes a disc recognition procedure immediately upon the insertion of a CD. The MediaMax driver reads two sectors of audio at a specific offset from the beginning of audio tracks—approximately 365 and 366 frames in (a CD frame stores $1/75$ second of sound). On unprotected discs, the software scans through every track in this way, but on MediaMax-protected albums, it stops after the first three tracks, apparently having detected an identifying feature. The software decides whether or not to block read access to the audio solely on the basis of information in this region, so we inferred that the identifying mechanism takes the form of an inaudible watermark embedded in this part of the audio stream.[6]

Locating the watermark amid megabytes of audio might have been difficult, but we had the advantage of a virtual Rosetta Stone. The actual Rosetta Stone—a 1500 lb. granite slab, unearthed in Rosetta, Egypt, in 1799—is inscribed with the same text written in three languages: ancient hieroglyphics, demotic (simplified) hieroglyphics, and Greek. Comparing these inscriptions provided the key to deciphering Egyptian hieroglyphic texts. Our Rosetta Stone was a single album, Velvet Revolver's *Contraband*, released in three different versions: a U.S. release protected by MediaMax, a European release protected by a passive scheme developed by Macrovision, and a Japanese release with no copy protection. We decoded the MediaMax watermark by examining the differences between the audio on these three discs. Binary comparison revealed no differences between the releases from Europe and Japan; however, the MediaMax-protected U.S. release differed slightly from the other two in certain parts of the recording. By carefully analyzing these differences—and repeatedly attempting to create new watermarked discs using the MediaMax active protection software as an oracle—we were able to deduce the structure of the watermark.

The MediaMax watermark is embedded in the audio of each track in 30 *clusters* of modified audio samples. Each cluster is made up of 288 marked 16-bit audio samples followed by 104 unaltered samples. Three mark clusters exactly fit into one 2352-byte CD audio frame. The watermark is centered at approximately frame 365 of the track; though the detection routine in the software only reads two frames, the mark extends several frames to either side of the designated read target to allow for imprecise seeking in the audio portion of the disc (a typical shortcoming of inexpensive CD drives). The MediaMax driver detects the watermark if at least one mark cluster is present in the region read by the detector.

A sequence of 288 bits that we call the *raw watermark* is embedded into the 288 marked audio samples of each mark cluster. A single bit of the raw watermark is embedded into an unmarked audio sample by setting one of the three least significant bits to the new bit value (as shown in bold below) and then setting the two other bits according to this table:[7]

Original bits	Marked bits					
	0__	**_0_**	**__0**	**1__**	**_1_**	**__1**
_____________111	**0**11	1**0**1	11**0**	**1**11	1**1**1	11**1**
_____________110	**0**11	1**0**1	11**0**	**1**10	1**1**0	11**1**
_____________101	**0**11	1**0**1	10**0**	**1**01	1**1**0	10**1**
_____________100	**0**11	1**0**0	10**0**	**1**00	1**1**0	10**1**
_____________011	**0**11	0**0**1	01**0**	**1**00	0**1**1	01**1**
_____________010	**0**10	0**0**1	01**0**	**1**00	0**1**0	01**1**
_____________001	**0**01	0**0**1	00**0**	**1**00	0**1**0	00**1**
_____________000	**0**00	0**0**0	00**0**	**1**00	0**1**0	00**1**

The position of the embedded bit in each sample follows a fixed sequence for every mark cluster. Each of the 288 bits is embedded in the first-, second-, or third-least-significant bit position of the sample according to this sequence:

```
2,3,1,1,2,2,3,3,2,3,3,3,1,3,2,3,2,1,3,2,2,3,2,2,
 2,1,3,3,2,1,2,3,3,1,2,2,3,1,2,3,3,1,1,2,2,1,1,3,
3,1,2,3,1,2,3,3,1,3,3,2,1,1,2,3,2,2,3,3,3,1,1,3,
 1,2,1,2,3,3,2,2,3,2,1,2,2,1,3,1,3,2,1,1,2,1,1,1,
2,3,2,1,1,2,3,2,1,3,2,2,2,3,1,2,1,3,3,3,3,1,1,1,
 2,1,1,2,2,2,2,3,1,2,3,2,1,3,1,2,2,3,1,1,3,1,1,1,
1,2,2,3,2,3,2,3,2,1,2,3,1,3,1,3,3,3,1,1,2,1,1,2,
 1,3,3,2,3,3,2,2,1,1,1,2,2,1,3,3,3,3,3,1,3,1,1,3,
2,2,3,1,2,1,2,3,3,2,1,1,3,2,1,1,2,2,1,3,3,2,2,3,
 1,3,2,2,2,3,1,1,1,1,3,2,1,3,1,1,2,2,3,2,3,1,1,2,
1,3,2,3,3,1,1,3,2,1,3,1,2,2,3,1,1,3,2,1,2,2,2,1,
3,3,1,2,3,3,3,1,2,2,3,1,2,3,1,1,3,2,2,1,3,2,1,3
```

The active protection software reads the raw watermark by reading the first, second, or third bit from each sample according to the sequence above. It determines whether the resulting 288-bit sequence is a valid watermark by checking certain properties of the sequence (represented below). It requires 96 positions in the sequence to have a fixed value, either 0 or 1. Another 192 positions are divided into 32 groups of linked values (denoted *a–z* and α–ζ below). In each group, three positions share the same value and three share the complement value. This allows the scheme to encode a 32-bit value (value A), though in the discs we studied it appears to take a different random value in each mark cluster of each protected title. The final 32 bits of the raw watermark may have arbitrary values (denoted by $\_$ below) and encode a second 32-bit value (value B). MediaMax version 5 uses this value to distinguish between original discs and backup copies burned through it proprietary player application.

$$\begin{gathered}
0, a, b, c, d, e, 0, 0, f, 0, g, 0, h, 0, i, d, j, \bar{j}, k, 0, l, m, 0, n, \\
o, p, \bar{e}, q, \bar{e}, r, 0, \bar{p}, s, d, \bar{m}, t, u, v, w, t, \bar{l}, a, x, c, u, 0, \bar{r}, l, \\
f, \bar{d}, v, 0, m, 0, \bar{q}, 0, y, c, z, 0, j, \bar{i}, \bar{g}, \alpha, \bar{s}, \bar{w}, \bar{h}, v, y, n, 0, 0, \\
\bar{h}, \bar{j}, \bar{u}, a, \beta, 0, \bar{v}, g, j, 0, 0, \bar{\beta}, \bar{i}, e, \bar{z}, 0, r, \gamma, \bar{a}, \delta, \bar{d}, \bar{z}, 0, \bar{v}, \\
\epsilon, 0, x, s, \bar{g}, \bar{r}, 0, \bar{b}, o, b, r, 0, y, \bar{\beta}, \bar{m}, h, 0, \bar{a}, n, \bar{f}, \bar{t}, 0, \bar{o}, 0, \\
\bar{\gamma}, \bar{\epsilon}, \bar{e}, 0, 0, \bar{k}, \bar{c}, \bar{x}, 0, \bar{f}, p, z, \bar{x}, i, 0, 0, \alpha, \bar{g}, 0, 1, w, \bar{t}, \bar{n}, \bar{w}, \\
i, 0, 0, \bar{j}, m, x, \beta, \bar{y}, \bar{p}, \bar{q}, 0, 0, 0, e, \bar{\beta}, 0, 0, 1, g, 0, p, l, 0, \bar{\alpha}, \\
t, h, \bar{d}, \bar{\epsilon}, \bar{w}, \gamma, \bar{\delta}, 0, \bar{p}, q, \bar{f}, 0, 1, \zeta, 0, \bar{c}, \zeta, \bar{\alpha}, \bar{s}, \bar{b}, \bar{\gamma}, \beta, 0, o, \\
0, q, \bar{i}, 0, 0, \bar{\alpha}, s, \epsilon, \bar{\epsilon}, \bar{h}, 0, \bar{k}, \bar{n}, \bar{\zeta}, \alpha, \bar{s}, \bar{z}, \bar{n}, \bar{c}, \bar{o}, \bar{b}, 0, \bar{t}, 0, \\
\bar{y}, \bar{v}, 0, \zeta, \bar{o}, 0, \bar{\zeta}, 0, u, \gamma, 0, \bar{y}, k, \bar{u}, z, \bar{\delta}, \bar{q}, k, \bar{r}, \bar{u}, \bar{\zeta}, \bar{\gamma}, \bar{l}, \bar{l}, \\
w, \bar{k}, \bar{a}, 0, \bar{\delta}, 0, \epsilon, \bar{m}, b, f, 0, 0, \bar{x}, \delta, \delta, 0, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \\
\_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_, \_
\end{gathered}$$

5.3 Attacks on the MediaMax Watermark

The MediaMax watermark fails to satisfy the indelibility and unforgeability requirements of an ideal disc recognition system. Far from being indelible, the mark is surprisingly brittle. Most advanced designs for robust audio watermarks [7, 6] manipulate the audio in the frequency domain and try to resist removal attempts that use lossy compression, multiple conversions between digital and analog formats, and other common transformations. In contrast, the MediaMax watermark is applied in the time domain and is rendered undetectable by even minor changes to the file. An adversary without any knowledge of the watermark's design could remove it by converting the tracks to a lossy format like MP3 and then burning them back to a CD, which can be accomplished easily with standard consumer applications. This would result in some minor loss of fidelity, but a more sophisticated adversary could prevent the mark from being detected with almost no degradation by flipping the least significant bit of one carefully chosen sample from each of the 30 watermark clusters, thereby preventing the mark from exhibiting the pattern required by the detector.

The watermark also fails to satisfy the unforgeability requirement. The mark's only defense against forgery is its complicated, unpublished design, but as is often the case this security by obscurity has proved tedious rather than impossible to defeat. As it turns out, an adversary needs only limited knowledge of the watermark—its location within a protected track and its confinement to the three least significant bits of each sample—to forge it with minimal loss of fidelity. Such an attacker could transplant the three least significant bits of each sample within the watermarked region of a protected track to the corresponding sample from an unprotected one. Transplanting these bits would cause distortion more audible that that caused by embedding the watermark since the copied bits are likely to differ by a greater amount from the original sample values; however, the damage to the audio quality would be limited since the marked region is only 0.4 seconds in duration. A more sophisticated adversary could apply a watermark to an unprotected track by deducing the full details of the structure of the watermark, as we did; she could then embed the mark in an arbitrary audio file just as well a licensed disc producer.

Though MediaMax did not do so, it is straightforward to create an unforgeable mark using digital signatures. The marking algorithm would extract a segment of music, compute its cryptographic hash, digitally sign the hash, and write the hash into the low-order bits of audio samples elsewhere in the music file. The recognition algorithm would recompute the hash, and extract and verify the signature. Though unforgeable, this mark would be no more indelible than the MediaMax scheme—making an indelible mark is a more difficult problem.

6 CD DRM Players

Increasingly, personal computers—and portable playback devices that attach to them—are users' primary means of organizing, transporting, and enjoying their music collections. Sony-BMG and its DRM vendors recognized this trend when they designed their copy protection technologies. Rather than inhibit all use with PCs, as some earlier anti-copying schemes did [10], XCP and MediaMax provide their own proprietary media players, shipped on each protected CD, that allow certain limited uses of the music subject to restrictions imposed by the copyright holder.[8]

The XCP and MediaMax players launch automatically using autorun when a protected disc is inserted into a PC. Both players have similar feature sets. They provide a rudimentary playback interface, allowing users to listen to protected albums, and they allow access to "bonus content," such as album art, liner notes, song lyrics, and links to artist web sites. The players access music on the disc, despite the active protection, by using a special back door interface provided by the active protection software.

XCP and MediaMax version 5 both permit users to burn copies of the entire album a limited number of times (typically three). These copies are created using a proprietary burning application integrated into the player. The copies include the player applications and the same active (and passive, for XCP) protection as the original album, but they do not allow any subsequent generations of copying.

Another feature of the player applications allows users to rip the tracks from the CD to their hard disks, but only in DRM-protected audio formats. Both schemes support the Windows Media Audio format by using a Microsoft product, the Windows Media Data Session Toolkit [17], to deliver DRM licenses that are bound to the PC where the files were ripped. The licenses allow the music to be transferred to portable devices that support Windows Media DRM or burned onto CDs, but the Windows Media files will not be usable if they are copied to another PC. Because XCP and MediaMax create Windows Media files, they are vulnerable to any attack that can defeat Windows Media DRM. Often, DRM interoperation allows attacks on one system to defeat other systems as well, because the attacker can transfer protected content into the system of her choice in order to extract it.

The XCP and MediaMax version 5 players both exhibit similar spyware-like behavior: phoning home to the vendor or record label with information about users' listening habits despite statements to the contrary from the vendors. Whenever a protected disc is inserted, the players contact web servers to retrieve images or banner ads to display. Part of the request is a code that identifies the album. XCP discs contact a Sony web site, `connected.sonymusic.com` [20]; MediaMax albums contact `license.sunncomm2.com`, a site operated by MediaMax's creator, SunnComm. These connections allow the servers to log the user's IP address, the date and time, and the identity of the album. This undisclosed data collection, in combination with other practices—installation without informed consent and the lack of an uninstaller—make XCP and MediaMax fit the consensus definition of spyware.

6.1 Attacks on Players

The XCP and MediaMax version 5 players were designed to enforce usage restrictions specified by content providers. In practice, they provide minimal security because there are many ways that users can bypass the limitations. Perhaps the most interesting class of attacks targets the limited number of burned copies permitted by the players. Both players are designed to enforce this limit without communicating with any networked server; thus, the player must keep track of how many allowed copies remain by storing state on the local machine.

It is well known that DRM systems like this are vulnerable to rollback attacks. A rollback attack backs up the state of the machine before performing the limited operation (in this case, burning the copy). When the operation is complete, the old system state is restored, and the DRM software is not able to determine that the operation has occurred. This kind of attack is easy to perform with virtual machine software like VMWare, which allows the entire state of the system to be saved or restored in a few clicks. XCP and MediaMax both fail under this attack, which allows unlimited copies to be burned with their players.

A refined variation of this attack targets only the specific pieces of state that the DRM system uses to remember the number of copies remaining. The XCP player uses a single file, `%windir%\system32\$sys$filesystem\$sys$parking`, to record how many copies remain for every XCP album that has been used on the system.[9] Rolling back this file after a disc copy operation would restore the original number of copies remaining.

A more advanced attacker can go further and modify the `$sys$parking` file to set the counter to an arbitrary value. The file consists of a 16 byte header followed by a series of 177 byte structures. For each XCP disc used on the machine, the file contains a whole-disc structure and an individual structure for each track. Each disc structure stores the number of permitted copies remaining for the disc as a 32-bit integer beginning 100 bytes from the start of the structure.

The file is protected by primitive encryption. Each structure is XORed with a repeating 256-bit pad. The pad—a single pad is used for all structures—is randomly chosen when XCP is first installed and stored in the system registry in the key `HKLM\SOFTWARE\$sys$reference\ClassID`. Note that this key, which is hidden by the rootkit, is intentionally misnamed "ClassID" to confuse investigators. Instead of a ClassID, it contains the 32 bytes of pad data.

Hiding the pad actually doesn't increase the security of the design. An attacker who knows only the format of the `$sys$parking` file and the current number of copies remaining can change the counter to an arbitrary value without needing to know the pad. Say the counter indicates that there are x copies remaining and the attacker wants to set it to y copies remaining. Without decrypting the structure, she can XOR the padded bytes where the counter is stored with the value $x \oplus y$. If the original value was padded with p, the new value will be $(x \oplus p) \oplus (x \oplus y) = (y \oplus p)$, y padded with p.

Ironically, Sony itself furnishes directions for carrying out another attack on the player DRM. Conspicuously absent from the XCP and MediaMax players is support for the Apple iPod—by far the most popular portable music player. A Sony FAQ blames Apple for this shortcoming and urges users to direct complaints to them: "Unfortunately, in order to directly and smoothly rip content into iTunes it [sic.] requires the assistance of Apple. To date, Apple has not been willing to cooperate with our protection vendors to make ripping to iTunes and to the iPod a simple experience." [23]. Strictly speaking, it is untrue that Sony requires Apple's cooperation to work with the iPod, as the iPod can import MP3s and other open formats. What Sony has difficulty doing is moving music to the iPod while keeping it wrapped in copy protection. This is because Apple has so far refused to support interoperation with its FairPlay DRM.

Yet so great is consumer demand for iPod compatibility that Sony gives out—to any customer who fills out a form on its web site [22]—instructions for working around its own copy protection and transforming the music into a DRM-free format that will work with the iPod. The procedure is simple but cumbersome: users are directed to use the player software to rip the songs into Windows Media DRM files; use Windows Media Player to burn the files to a blank CD, which will be free of copy protection; and then use iTunes to rip the songs once more and transfer them to the iPod.

6.2 MediaMax Player Security Risks

Besides suffering from several kinds of attacks that expose the music content to copying, the MediaMax version 5 player makes the user's system more vulnerable to attack. When a MediaMax CD is inserted into a computer, Windows autorun launches an installer from the disc. Even before displaying a license agreement, MediaMax copies almost twelve megabytes of files and data related to the MediaMax player to the hard disk. Jesse Burns and Alex Stamos of iSEC Partners discovered that the MediaMax installer sets file permissions that allow any user to modify its code directory and the files and programs in it [5].

As Burns and Stamos realized, the lax permissions allow a non-privileged user to replace the executable code in the MediaMax player files with malicious code. The next time a user plays a MediaMax-protected CD, the attack code will be executed with that user's security privileges. The MediaMax player requires Power User or Administrator privileges to run, so it's likely that the attacker's code will run with almost complete control of the system.

Normally, this problem could be fixed by manually correcting the errant permissions. However, MediaMax aggressively updates the installed player code each time the software on a protected disc autoruns or is launched manually. As part of this update, the permissions on the installation directory are reset to the insecure state.

We discovered a variation of the attack suggested by Burns and Stamos that allows the attack code to be installed even if the user has never consented to the installation of MediaMax, and to be triggered immediately whenever the user inserts a MediaMax CD. In our attack, the attacker places hostile code in the `DllMain` procedure of a code file called `MediaMax.dll`, which MediaMax installs even before displaying the EULA. The next time a MediaMax CD is inserted, the installer autoruns and immediately attempts to check the version of the installed `MediaMax.dll` file. To do this, the installer calls the Windows `LoadLibrary` function on the DLL file, which causes the file's `DllMain` procedure to execute, along with any attack code placed there.

This problem is exacerbated because parts of the MediaMax software are installed automatically and without consent. Users who have declined the EULA likely assume that MediaMax has not been installed, and so most will be unaware that they are vulnerable. The same installer code performs the dangerous version check as soon as the CD is inserted. A CD that prompted the user to accept a license before installing code would give the user a chance to head off the attack.

Fixing this problem permanently without losing the use of protected discs requires installing a patch from SunnComm. Unfortunately, as we discovered, the initial patch released by Sony-BMG in response to the iSEC report was capable of triggering precisely the kind of attack it was supposed to prevent. In the process of updating MediaMax, the patch checked the version of `MediaMax.dll` just like the MediaMax installer does. If this file was already modified by an attacker, the process of applying the security patch would execute the attack code. Prior versions of the MediaMax uninstaller had the same vulnerability, though both the uninstaller and the patch have since been replaced with versions that do not suffer from this problem.

7 Deactivation

Active protection methods install and run software components that interfere with accesses to a CD. Users can remove or deactivate the active protection software by using standard system administration tools that are designed to find, characterize, and control the programs installed on a machine. Deactivating the protection will enable arbitrary use or ripping of the music, and it is difficult to stop if the user has system administrator privileges. In this section, we discuss how active protection may be deactivated.

7.1 Deactivating MediaMax

The MediaMax active protection software is easy to deactivate, being comprised of a single device driver named `sbcphid`. The driver can be removed by using the Windows command `sc delete sbcphid` to stop the driver, and then removing the `sbcphid.sys` file containing the driver code. MediaMax-protected albums can then be accessed freely.

7.2 Defenses Against Deactivation

To counter deactivation attempts, a vendor might try technical tricks to evade detection and frustrate removal of the active protection software. An example is the rootkit-like behavior of XCP, discovered by Mark Russinovich [21]. When XCP installs its active protection software, it also installs a second program—the rootkit—that conceals any file, process, or registry key whose name begins with the prefix `$sys$`. The result is that XCP's main installation directory, and most of its registry keys, files, and processes, become invisible to normal programs and administration tools.

The rootkit is a kernel-level driver named `$sys$aries` that is set to automatically load early in the boot process. When the rootkit starts, it hooks several Windows system calls by modifying the system service dispatch table (the kernel's `KeServiceDescriptorTable` structure) which is an array of pointers to the kernel functions that implement basic system calls. The rootkit modifies the behavior of four system calls: `NtQueryDirectoryFile`, `NtCreateFile`, `NtQuerySystemInformation`, and `NtEnumerateKey`.[10] These calls are used to enumerate files, processes, and registry entries. The rootkit filters the data returned by these calls to hide items whose names begin with `$sys$`.

On intercepting a function call, the rootkit checks the name of the calling process. If the name of the calling process begins with `$sys$`, the rootkit returns the results of the real kernel function without alteration so that XCP's own processes have an accurate view of the system.

The XCP rootkit increases users' vulnerability to attack by allowing any software to hide—not just XCP. Malware authors can exploit the fact that any files, registry keys, or processes with names beginning in `$sys$` will be hidden, thereby saving the trouble of installing their own rootkits. Malware that lacks the privileges to install its own rootkit can still rely on XCP's rootkit.

Only kernel-level processes can patch the Windows system service dispatch table, and only privileged users—normally, members of the Administrators or Power Users groups—can install such processes. (XCP itself requires these privileges to install.) Malicious code running as an unprivileged user can't normally install a rootkit that intercepts system calls. But if the XCP rootkit is installed, it will hide all programs that adopt the `$sys$` prefix so that even privileged users will be unable to see them. This vulnerability has already been exploited by at least two Trojan horses seen in the wild [15, 14].

The rootkit opens at least one more security vulnerability. The modified functions do not check for errors as carefully as the original Windows functions do, so the rootkit makes it possible for an ordinary program to crash the system by calling one of the hooked functions, for example by calling `NtCreateFile` with an invalid `ObjectAttributes` argument. We do not believe this vulnerability can be exploited to run arbitrary code.

7.3 Deactivating XCP

Deactivating XCP's active protection is more complicated because it comprises several processes that are more deeply entangled in the system configuration, and are hidden by the XCP rootkit. Deactivation requires a three-step procedure.

The first step is to deactivate and remove the rootkit, by the same procedure used to deactivate MediaMax (except that the driver's name is `aries.sys`). Disabling the rootkit and then rebooting exposes the previously hidden files, registry entries, and processes.

The second step is to edit the registry to remove references to XCP's filter drivers and CoDeviceInstallers. XCP uses the Windows filter driver facility to intercept commands to the CD drives and IDE bus. If the code for these filter drivers is removed but the entries pointing to that code are not removed from the registry, the CD and IDE device drivers will fail to initialize. This can cause the CD drives to malfunction, or, worse, can stop the system from booting if the IDE device driver is disabled. The registry entries can be eliminated by removing any reference to a driver named `$sys$cor` from any registry entries named `UpperDrivers` or `LowerDrivers`, and removing any lines containing `$sys$caj` from any list of CoDeviceInstallers in the registry.

The third step is to delete the XCP services and remove the XCP program files. Services named `$sys$lim`, `$sys$oct`, `$sys$drmserver`, `cd_proxy`, and `$sys$cor` can be deactivated using the `sc delete` command, and then files named `crater.sys`, `lim.sys`, `oct.sys`, `$sys$cor.sys`, `$sys$caj.dll`, and `$sys$upgtool.exe` can be deleted. After rebooting, the two remaining files named `CDProxyServ.exe` and `$sys$DRMServer.exe` can be removed.

Performing these steps will deactivate the XCP active protection, leaving only the passive protection on XCP CDs in force. The procedure easily could be automated to create a point-and-click removal tool.

7.4 Impact of Spyware Tactics

The use of rootkits and other spyware tactics harms users by undermining their ability to manage their computers. If users lose effective control over which programs run on their computers, they can no longer patch malfunctioning programs or remove unneeded programs. Managing a system securely is difficult enough without spyware tactics making it even harder.

Though it is no surprise that spyware tactics would be attractive to DRM designers, it is a bit surprising that mass-market DRM vendors chose to use those tactics despite their impact on users. If only one vendor had chosen to use such tactics, we could write it off as an aberration. But two vendors made that choice, which is probably not a coincidence. We suspect that the vendors let the lure of platform building override the risk to users.

7.5 Summary of Deactivation Attacks

Ultimately, there is little a CD DRM vendor can do to stop users from deactivating active protection software. Vendors' attempts to frustrate users' control of their machines are harmful and will trigger a strong backlash from users. In practice, vendors will probably have to provide some kind of uninstaller—users will insist on it, and some users will need it to deal with the bugs and incompatibilities that crop up inevitably in complex software. Once an uninstaller is released, users can use it to remove the DRM software. Determined users will be able to keep CD DRM software off of their machines.

8 Uninstallation

The DRM vendors responded to user complaints about spyware-like behavior by offering uninstallers that would remove their software from users' systems. Uninstallers had been available before but were very difficult to acquire. For example, to get the original XCP uninstaller, a user had to fill out an online form involving personal information, then wait a few days for a reply email, then fill out another online form and install some software, then wait a few days for yet another email, and finally click a URL in the last email. It is hard to explain the complexity of this procedure, except as a way to deter users from uninstalling XCP.

The uninstallers, when users did manage to get them, did not behave like ordinary software uninstallers. Normal uninstallers are programs that can be acquired and used by any user who has the software. The first XCP uninstaller was customized for each user so that it would only work for a limited time and only on the computer on which the user had filled out the second form. This meant, for example, that if a user uninstalled XCP but it was reinstalled later—say, if the user inserted an XCP CD—the user could not use the same uninstaller again but would have to go through the entire process again to request a new one.

Customizing the uninstaller is more difficult, compared to a traditional uninstaller, for both vendor and user, so it must benefit the vendor somehow. One benefit is to the vendor's platform building strategy, which takes a step backward every time a user uninstalls the software. Customizing the uninstaller allows the vendor to control who receives the uninstaller and to change the terms under which it is delivered.

As user complaints mounted, Sony-BMG announced that unrestricted uninstallers for both XCP and MediaMax would be released from the vendors' web sites. Both vendors chose to make these uninstallers available as ActiveX controls. By an unfortunate coincidence, both uninstallers turned out to open the same serious vulnerability on any computer where they were used.

8.1 MediaMax Uninstaller Vulnerability

The original MediaMax uninstaller uses a proprietary ActiveX control, `AxWebRemove.ocx`, created and signed by SunnComm. Users visiting the MediaMax uninstaller web page are prompted to install the control, then the web page uninstalls MediaMax by invoking one of the control's methods.

This method, `Remove`, takes a URL and a numeric key as arguments. `Remove` contacts the URL, passing it the key. If the server finds the key to be valid, it returns another URL for the uninstaller. The ActiveX control downloads code from the uninstaller URL and then executes it. After running the uninstaller, the ActiveX control contacts the server again to notify it that the key had been used. MediaMax has been removed, but the ActiveX control remains on the user's system.

At this point, a malicious attacker's web page can invoke the control's `Remove` method, passing it a URL pointing to a malicious server controlled by the attacker. The control could contact this server, and then download and run code from a location supplied by the malicious server. By this method, an adversary could run arbitrary code on the user's system.

The flaw in this design, of course, is that MediaMax ActiveX control does not validate the URL it is passed, and does not validate the downloaded code before running it. Validating these items, perhaps using digital signatures, would have eliminated the vulnerability.

8.2 XCP Uninstaller Vulnerability

The original XCP uninstaller contains the same design flaw and is only slightly more difficult to exploit. XCP's ActiveX-based uninstaller invokes a proprietary ActiveX control named `CodeSupport.ocx`. Usually this control is installed in the second step of the three-step XCP uninstall process. In this step, a pseudorandom code generated by the ActiveX control is sent to the XCP server. The same code is written to the system registry. Eventually the user receives an email with a link to another web page that uses the ActiveX control to remove XCP, but only after verifying that the correct code is in the registry on the local system. This check tethers the uninstaller to the machine from which the uninstallation request was made. Due to this design, the vulnerable control may be present on a user's system even if she never performed the step in the uninstallation process where XCP is removed.

Matti Nikki first noted that the XCP ActiveX control contains suspiciously-named methods, including `InstallUpdate(url)`, `Uninstall(url)`, and `RebootMachine()` [18]. He demonstrated that the control was still present after the XCP uninstallation was complete, and that its methods (including one that rebooted the computer) were scriptable from any web page without further browser security warnings.

We found that the `InstallUpdate` and `Uninstall` methods have an even more serious flaw. Each takes as an argument a URL pointing to a specially formatted archive that contains updater or uninstaller code and data files. When these methods are invoked, the archive is retrieved from the provided URL and stored in a temporary location. For the `InstallUpdate` method, the ActiveX control extracts from the archive a file named `InstallLite.dll` and calls a function in this DLL named `InstallXCP`.

Like the MediaMax ActiveX control, the XCP control does not validate the download URL or the downloaded archive. The only barrier to using the control to execute arbitrary code is the proprietary format of the archive file. We determined the format by disassembling the control. The archive file consists of several blocks of gzip-compressed data, each storing a separate file and preceded with a short header. At the end of the archive, a catalog structure lists metadata for each of the blocks, including a 32-bit CRC. The control verifies this CRC before executing code from the DLL.

With knowledge of this file format, we were able to construct an archive containing (benign proof-of-concept) exploit code, and a web page that would install and run our code on a user's system without any browser security warnings, on a computer containing the XCP control. The same method would allow a malicious web site to execute arbitrary code on the user's machine. Like the MediaMax uninstaller flaw, this problem is especially dangerous because users who have completed the uninstallation may not be aware that they are still vulnerable.

Obviously, these vulnerabilities could have been prevented by careful design and programming. But they were only possible at all because the vendors chose to deliver the uninstallers via this ActiveX method rather than using an ordinary download. We conjecture that the vendors made this choice because they wanted to retain the ability to rewrite, modify, or cancel the uninstaller later, in order to further their platform building strategies.

9 Compatibility and Software Updates

Compared to other media on which software is distributed, compact discs have a very long life. Many compact discs will still be inserted into computers and other players twenty years or more after they are first bought. If a particular version of DRM software is shipped on a new CD, that software version may well try to install and run decades after it was developed. The same is not true of most software, even when shipped on a CD-ROM. Very few if any of today's Windows XP CDs will be inserted into computers in 2026; but today's music CDs will be, so their DRM software must be designed carefully for future compatibility.

The software should be designed for *safety*, so as not to cause crashes or malfunction of other software, and may be designed for *efficacy*, to ensure that its anti-copying features remain effective.

9.1 Supporting Safety by Deactivating Old Software

Safety is easier to achieve, and probably more important. One approach is to design the DRM software to be inert and harmless on future systems. Both XCP and MediaMax do this by relying on Windows autorun, which is likely to be disabled in future versions of Windows for security reasons. If the upcoming Windows Vista disables autorun by default, XCP and MediaMax will be inert on most Vista systems. Perhaps XCP and MediaMax used autorun for safety reasons; but more likely, this choice was expedient for other reasons.

Another safety technique is to build in a sunset date after which the software will make itself inert. A sunset would improve safety but would have relatively little effect on record label revenue for most discs, as we expect nearly all revenue from the disc to have been extracted from the customer in the first three years after she buys it. If in the future more copies of the album are pressed, these could have updated DRM software with a later sunset.

9.2 Updating the Software

When a new version of DRM software is released, it can be shipped on newly pressed CDs, but existing CDs cannot be modified retroactively. Updates for existing users can be delivered either by download or on new CDs. Downloads are faster but require an Internet connection; CD delivery is slower but can reach non-networked machines.

Users will generally cooperate with updates that help them by improving safety or making the software more useful. But updates to retain the efficacy of the software's usage controls will not be welcomed by users.

Users have many ways to stop updates from downloading or installing, such as write-protecting the software's code so that it cannot be updated, or using a personal firewall to block network connections to the vendor's download servers. System security tools, which are designed generally to stop unwanted network connections, downloads, and code installation, can be set to treat CD DRM software as malware.

A DRM vendor who wants to deliver unwanted updates has two options. First, the vendor can simply offer updates and hope some users will not bother to block them. For the vendor and record label, this is better than nothing. Alternatively, the vendor can try to force users to accept updates.

9.3 Forcing Updates

If a user has the ability to block DRM software updates, a vendor who wants an update must somehow convince the user that updating is in her best interest. One approach is to make a non-updated system painful to use.

Ruling out dangerous and legally risky tactics such as logic bombs that destroy the user's system or hold her (unrelated) data hostage, the vendor's strongest tactic for forcing updates is to make the DRM software block all access to protected CDs until the user accepts an update. The DRM software might check with a network server, which periodically would produce a digitally signed and dated certificate listing allowed versions of the DRM software. If the software on the user's system found that its version number was not on the list (or if it could not get a recent list), it would block all access to protected discs. The user would then have to update to a new version to get access to her protected CDs.

This approach would convince some users to update, and would thereby prolong the DRM's efficacy for those users. But it has several drawbacks. If the computer is not networked, the software will eventually lock down because it cannot get certificates. (If the software kept working in this case, users could avoid updates by preventing the DRM software from making network connections.) A bug in the software could cause an accidental but irreversible lockdown. Or the software could lock itself down if the vendor's Internet site is shut down, for example if the vendor goes bankrupt.

Strong-arm tactics can also be counterproductive, by giving the user further reason to defeat or remove the DRM software.[11] The software is more likely to remain on the user's system if it does not behave annoyingly. Trying to force updates can reduce the DRM system's efficacy if it convinces users to remove the DRM altogether. From the user's standpoint, every software update is a security risk—a possible vector for hostile or buggy code.

Given the problems with forced updates, and the user backlash they likely would have triggered, we are not surprised that neither XCP nor MediaMax tried to force updates.

10 User Outrage, and the Fight to Control Users' Computers

One notable aspect of the Sony CD DRM episode was the level of outrage expressed by users. All too frequently, bugs in popular software products endanger users' security or privacy, and users just grumble and update their software. Users' anger over the CD DRM episode was much more intense. What made this issue so different?

There are three answers. First, many users did not expect audio CDs to contain software. Users did not want the software, and they recognized that Sony-BMG chose to include it anyway. Unlike (say) an email client, which necessarily includes complex software components that might have bugs, CDs need not include software, so users are less willing to accept the risk of security problems in order to get CDs.

Second, some harmful aspects of the CD DRM software reflected deliberate choices by the vendors (and by extension, Sony-BMG). Users who might be willing to forgive implementation errors will not accept the deliberate introduction of security and privacy risks. There can be little question that XCP's rootkit functionality, the installation without consent of MediaMax software, the lack of uninstallers, and phone-home behavior were put in place deliberately by the vendors.

Third, when the vendors did make apparent implementation errors, the errors were compounded by the products' aggressive installation and reluctant uninstallation mechanisms. For example, the file permission problem discovered by Burns and Stamos was difficult to fix because the MediaMax autorun program aggressively reset the permissions to dangerous values, without asking the user for permission, every time a disc was inserted. Similarly, the vendors' apparent desire to limit use of their uninstallers led to designs that relied on downloading code using ActiveX controls—leaving users just one bug away from critical code-download vulnerabilities.

These factors led some users to conclude that Sony-BMG and the DRM vendors not only put their own business interests ahead of their customers' interests, but also made deliberate choices that endangered customers' security and privacy. Users who would have forgiven a few implementation mistakes by a well-intentioned vendor were not so quick to forgive when they felt the vulnerabilities were less than accidental.

Though Sony-BMG and other copyright owners will presumably tread more carefully in the future, there remains a fundamental tension between DRM vendors' desire to control and limit how computers are used, and the need of users to manage their own systems. Users and DRM distributors will continue to struggle for control of users' computers.

11 Conclusion

Our analysis of Sony-BMG's CD DRM carries wider lessons for content companies, DRM vendors, policymakers, end users, and the security community. We draw six main conclusions.

First, the design of DRM systems is driven strongly by the incentives of the content distributor and the DRM vendor, but these incentives are not always aligned. Where they differ, the DRM design will not necessarily serve the interests of copyright owners, not to mention artists.

Second, DRM, even if backed by a major content distributor, can expose users to significant security and privacy risks. Incentives for aggressive platform building drive vendors toward spyware tactics that exacerbate these risks.

Third, there can be an inverse relation between the efficacy of DRM and the user's ability to defend her computer from unrelated security and privacy risks. The user's best defense is rooted in understanding and controlling which software is installed, but many DRM systems rely on undermining this understanding and control.

Fourth, CD DRM systems are mostly ineffective at controlling uses of content. Major increases in complexity have not increased their effectiveness over that of early schemes, and may in fact have made things worse by creating more avenues for attack. We think it unlikely that future CD DRM systems will do better.

Fifth, the design of DRM systems is only weakly connected to the contours of copyright law. The systems make no pretense of enforcing copyright law as written, but instead seek to enforce rules dictated by the label's and vendor's business models. These rules, and the technologies that try to enforce them, implicate other public policy concerns, such as privacy and security.

Finally, the stakes are high. Bad DRM design choices can seriously harm users, create major liability for copyright owners and DRM vendors, and ultimately reduce artists' incentive to create.

Acknowledgments

We are grateful for the expert legal advice of Deirdre Mulligan and her colleagues at U.C. Berkeley: Aaron Perzanowski, Sara Adibisedeh, Azra Medjedovic, Brian W. Carver, Jack Lerner, and Joseph Lorenzo Hall. We are also grateful to Clayton Marsh at Princeton. Sadly, research of this type does seem to require support from a team of lawyers.

We thank the readers of Freedom to Tinker for their comments on partial drafts that we posted there; thanks especially to C. Scott Ananian, Randall Chertkow, Tim Howland, Edward Kuns, Jim Lyon, Tobias Robison, Adam Shostack, Ned Ulbricht, and several pseudonymous commenters. Jeff Dwoskin provided valuable technical assistance, and Shirley Gaw, Janek Klawe, and Harlan Yu gave helpful feedback. We are also grateful to the anonymous reviewers for their suggestions. Thanks to Claire Felten for help with copy editing.

This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Notes

[1]As news of the rootkit spread, we added to the public discussion with a series of 27 blog posts analyzing XCP and MediaMax. This paper provides a more systematic analysis, along with much new information. Our original blog entries can be read at http://www.freedom-to-tinker.com/?cat=30&m=2005.

[2]Music industry *rhetoric* about DRM often focuses on P2P, and some in the industry probably still think that DRM can stop P2P sharing. We believe that industry decision makers know otherwise. The design of the systems we studied in this paper supports this view.

[3]Similar application blacklisting techniques have been used in other security contexts. The client software for World of Warcraft, a massively multiplayer online role playing game, checks running applications against a regularly updated blacklist of programs used to cheat in the game [12].

[4]An extreme extension of this would be to adopt rootkit-like techniques to conceal the copying application's presence, just as XCP hides its active protection software.

[5]Forging a mark is probably not copyright infringement. Unlike the musical work in which it is embedded, the mark itself is functional and contains little or no expression, and therefore seems unlikely to qualify for copyright protection. In principle, the mark recognition process could be covered by a patent, but we are unaware of any such patent relating to XCP or MediaMax. Even if the vendor does have a legal remedy, it seems worthwhile to design the mark to prevent forgery if the cost of doing so is low.

[6]By locating the watermark nearly five seconds after the start of the track rather than at the very beginning, MediaMax reduces the likelihood that it will occur in a very quiet passage (where it might be more audible) and makes cropping it out more destructive.

[7]This design seems to be intended to lessen the audible distortion caused by setting one of the bits to the watermark value. The change in the other two bits reduces the magnitude of the difference from the

original audio sample, but it also introduces a highly uneven distribution in the three least significant bits that makes the watermark easier to detect or remove.

[8]The restrictions imposed by the DRM players only loosely track the contours of copyright law. Some uses that could be prohibited under copyright—such as burning three copies to give to friends—are allowed by the software, while some perfectly legal uses—like transferring the music to one's iPod—are prevented.

[9]This file is hidden and protected by the XCP rootkit. Before the user can access the file, the rootkit must be disabled, as described in Section 7.2. We did not determine how the MediaMax player stores the number of copies remaining.

[10]The rootkit also hooks `NtOpenKey` but does not alter its behavior.

[11]Users could also mislead the DRM software about the date and time, but most users with the inclination to do that would probably just remove the DRM software altogether.

References

[1] Class action complaint. In *Hull et al. v. Sony BMG et al.*, 2005. http://www.eff.org/IP/DRM/Sony-BMG/sony_complaint.pdf.

[2] Consolidated amended class action complaint. In *Michaelson et al. v. Sony BMG et al.*, 2005. http://sonysuit.com/classactions/michaelson/15.pdf.

[3] Original plantiff's petition. In *State of Texas v. Sony BMG Music Entertainment*, 2005. http://www.oag.state.tx.us/newspubs/releases/2005/112105sony_pop.pdf.

[4] Peter Biddle, Paul England, Marcus Peinado, and Bryan Willman. The Darknet and the future of content distribution. In *ACM Workshop on Digital Rights Management*, November 2002.

[5] Jesse Burns and Alex Stamos. Media Max access control vulnerability, November 2005. http://www.eff.org/IP/DRM/Sony-BMG/MediaMaxVulnerabilityReport.pdf.

[6] Ingemar Cox, Joe Kilian, Tom Leighton, and Talal Shamoon. Secure spread spectrum watermarking for multimedia. *IEEE Transactions on Image Processing*, 6(12):1673–1687, 1997.

[7] Scott A. Craver, Min Wu, Bede Liu, Adam Stubblefield, Ben Swartzlander, Dan S. Wallach, Drew Dean, and Edward W. Felten. Reading between the lines: Lessons from the SDMI challenge. In *Proc. 10th USENIX Security Symposium*, August 2001.

[8] Edward W. Felten and J. Alex Halderman. Digital rights management, spyware, and security. *IEEE Security and Privacy*, January/February 2006.

[9] Allan Friedman, Roshan Baliga, Deb Dasgupta, and Anna Dreyer. Understanding the broadcast flag: a threat analysis model. In *Telecommunications Policy*, volume 28, pages 503–521, 2004.

[10] J. Alex Halderman. Evaluating new copy-prevention techniques for audio CDs. In *Proc. ACM Workshop on Digital Rights Management (DRM)*, Washington, D.C., November 2002.

[11] J. Alex Halderman. Analysis of the MediaMax CD3 copy-prevention system. Technical Report TR-679-03, Princeton University Computer Science Department, Princeton, New Jersey, 2003.

[12] Greg Hoglund. 4.5 million copies of EULA-compliant spyware, October 2005. http://www.rootkit.com/blog.php?newsid=358.

[13] Greg Hoglund and James Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley, 2005.

[14] Kazumasa Itabashi. Trojan.Welomoch technical description, December 2005. http://securityresponse.symantec.com/avcenter/venc/data/trojan.welomoch.html.

[15] Yana Liu. Backdoor.Ryknos.B technical description, November 2005. http://securityresponse.symantec.com/avcenter/venc/data/backdoor.ryknos.b.html.

[16] MediaMax Technology Corp. Annual report (S.E.C. Form 10-KSB/A), September 2005.

[17] Microsoft Corporation. Windows Media data session toolkit. http://download.microsoft/com/download/a/1/a/a1a66a2c-f5f1-450a-979b-ddf790756f1d/Data_Session_Datasheet.pdf.

[18] Matti Nikki. Muzzy's research about Sony's XCP DRM system, December 2005. http://hack.fi/~muzzy/sony-drm/.

[19] K. Reichert and G. Troitsch. Kopierschutz mit filzstift knacken. *Chip.de*, May 2002.

[20] Mark Russinovich. More on Sony: Dangerous decloaking patch, EULAs and phoning home, November 2005. http://www.sysinternals.com/blog/2005/11/more-on-sony-dangerous-decloaking.htm.

[21] Mark Russinovich. Sony, rootkits and digital rights management gone too far, October 2005. http://www.sysinternals.com/blog/2005/10/sony-rootkits-and-digital-rights.html.

[22] Sony-BMG Music Entertainment. Portable device: iPod information. http://cp.sonybmg.com/xcp/english/form10.html.

[23] Sony-BMG Music Entertainment. XCP frequently asked questions. http://cp.sonybmg.com/xcp/english/faq.html.

Milk or Wine: Does Software Security Improve with Age? [*†]

Andy Ozment
MIT Lincoln Laboratory[‡]

Stuart E. Schechter
MIT Lincoln Laboratory

Abstract

We examine the code base of the OpenBSD operating system to determine whether its security is increasing over time. We measure the rate at which new code has been introduced and the rate at which vulnerabilities have been reported over the last 7.5 years and fifteen versions.

We learn that 61% of the lines of code in today's OpenBSD are *foundational*: they were introduced prior to the release of the initial version we studied and have not been altered since. We also learn that 62% of reported vulnerabilities were present when the study began and can also be considered to be foundational.

We find strong statistical evidence of a decrease in the rate at which foundational vulnerabilities are being reported. However, this decrease is anything but brisk: foundational vulnerabilities have a median lifetime of *at least* 2.6 years.

Finally, we examined the density of vulnerabilities in the code that was altered/introduced in each version. The densities ranged from 0 to 0.033 vulnerabilities reported per thousand lines of code. These densities will increase as more vulnerabilities are reported.

[*]This work is sponsored by the I3P under Air Force Contract FA8721-05-0002. Opinions, interpretations, conclusions and recommendations are those of the author(s) and are not necessarily endorsed by the United States Government.

[†]This work was produced under the auspices of the Institute for Information Infrastructure Protection (I3P) research program. The I3P is managed by Dartmouth College, and supported under Award number 2003-TK-TX-0003 from the U.S. Department of Homeland Security, Science and Technology Directorate. Points of view in this document are those of the authors and do not necessarily represent the official position of the U.S. Department of Homeland Security, the Science and Technology Directorate, the I3P, or Dartmouth College.

[‡]Currently at the University of Cambridge

1 Introduction

Many in the security research community have criticized both the insecurity of software products and developers' perceived inattention to security. However, we have lacked quantitative evidence that such attention can improve a product's security over time. Seeking such evidence, we asked whether efforts by the OpenBSD development team to secure their product have decreased the rate at which vulnerabilities are reported.

In particular, we are interested in responding to the work of Eric Rescorla [11]. He used data from ICAT[1] to argue that the rate at which vulnerabilities are reported has not decreased with time; however, limitations in the data he used prompted us to investigate this area further.

We chose OpenBSD version 2.3 as our *foundation version*, and we collected 7.5 years of data on the vulnerabilities reported in OpenBSD since that version's release. In particular, we focused our analysis on *foundational vulnerabilities*: those introduced prior to the release of the foundation version. We also analyzed the evolution of the code base. We were driven by the goal of answering the following questions:

1. How much does legacy code influence security today?
2. Do larger code changes have more vulnerabilities?
3. Do today's coders introduce fewer vulnerabilities per line of code?
4. What is the median lifetime of a vulnerability?

Most importantly:

5. Has there been a decline in the rate at which foundational vulnerabilities in OpenBSD are reported?

In the upcoming section, we discuss the limitations of vulnerability reporting data; these limitations may result in our analysis underestimating increases in the security of OpenBSD. In Section 3, we elaborate on the methodology used to collect the data sets employed in this work. We then describe the results of our source code and vulnerability density analysis in Section 4. In Section 5 we provide statistical evidence that the rate at which foundational vulnerabilities are reported is decreasing. Finally, we discuss related literature in Section 6 and conclude in Section 7.

2 Limitations of vulnerability analyses

Our analysis uses the rate of vulnerability reports to measure one characteristic of OpenBSD's security. We use this data to estimate the size of the remaining pool of unreported vulnerabilities and to estimate the expected frequency with which new vulnerabilities will be reported. However, this information is only one aspect of the security of OpenBSD. The OpenBSD development team has not only worked to increase the security of the system's code base; they have also worked to improve its overall security architecture. These improvements include new security functionality and safeguards that reduce the severity of vulnerabilities.

2.1 New security functionality

The addition of valuable new security functionality, like OpenSSH, increases the amount of code that is deemed security-critical and may thus increase the pool of reportable vulnerabilities. This increase does not necessarily imply that the code is less secure: it may only mean that the operating system has assumed new security responsibilities.

2.2 Reductions in vulnerability severity

Architectural improvements that reduce the severity of a vulnerability—but do not eliminate it entirely—can improve security without reducing the rate at which vulnerabilities are discovered and reported. For example, the OpenBSD team improved the security architecture of OpenBSD by adding stack-guarding tools and randomized memory allocation [4], both of which reduce the severity of vulnerabilities within the code base.

These security improvements are not accounted for in our study, because we lack an accurate and unbiased methodology with which to assess the severity of vulnerabilities. Simply measuring reductions in the total pool of vulnerabilities is thus likely to underestimate improvements to the security of the overall system.

2.3 The influence of effort & skill on vulnerability discovery

The rate at which vulnerabilities are discovered and reported depends on the level of effort being expended to do so. To measure how much more difficult it has become to find vulnerabilities over time, we would need to normalize the rate of discovery by the effort being exerted and the skills of those exerting it.

Unfortunately, vulnerability reports do not include estimates of how many individuals were involved in examining the software, the time they spent, or their relative skills.

3 Methodology

We chose to study OpenBSD because its developers have long prioritized security [8]. In his work, Rescorla found no convincing evidence for a decrease in the rate of vulnerability reporting for three operating systems: Windows NT 4.0, Solaris 2.5.1, and FreeBSD 4.0 [11]. He did find a decrease in the reporting rate for RedHat 6.2, but he notes the existence of confounding factors for that system. We therefore sought to test a system whose developers focused on finding and removing vulnerabilities: if we had replicated Rescorla's results with this system, then less security-focused systems would presumably have the same results. Another reason that we selected OpenBSD is that its entire source code and every change that has been made to it are readily accessible via a publicly accessible CVS repository.

The initial release of OpenBSD was version 2.0; this version was forked from NetBSD 1.1 in late 1996. Prior to version 2.2, the OpenBSD developers performed an extensive security audit and repaired numerous vulnerabilities without reporting them. In version 2.3, the OpenBSD team changed the way they integrated X11 into the code base. We therefore selected version 2.3, released on 19 May 1998, as the earliest version for our data set: it was the first truly stable release in which vulnerabilities were consistently documented. We refer to this version as the foundation version, and we refer to code and vulnerabilities present before the release of this version as foundational code and foundational vulnerabilities.

The OpenBSD project releases a new version approximately every six months, incrementing the version number by 0.1. Our study incorporates the fifteen versions of OpenBSD from 2.3 to 3.7, inclusive.

3.1 The vulnerability data set

The OpenBSD vulnerability data set was created through the following process:

	Version in which the vulnerability was born															
Version in which the vulnerability died	2.3	2.4	2.5	2.6	2.7	2.8	2.9	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7	Total
2.3	5															5
2.4	11	0														11
2.5	6	0	1													7
2.6	5	1	0	0												6
2.7	12	4	2	2	2											22
2.8	12	1	0	1	2	0										16
2.9	4	0	0	2	0	0	0									6
3.0	3	1	0	0	1	0	2	0								7
3.1	8	2	1	2	0	0	0	1	1							15
3.2	6	2	0	0	0	0	1	2	0	1						12
3.3	2	1	0	2	0	0	0	0	0	0	2					7
3.4	2	0	0	0	1	0	1	0	1	0	0	0				5
3.5	7	1	1	0	0	0	0	2	0	1	0	0	1			13
3.6	3	0	1	0	0	0	0	0	0	0	0	0	0	0		4
3.7	1	1	0	0	1	0	0	0	0	0	0	0	1	0	0	4
Total	87	14	6	9	7	0	4	5	2	2	2	0	2	0	0	140
MLOC	10.1	0.4	0.3	1.1	0.8	0.4	2.2	0.6	0.8	0.3	0.3	0.8	1.4	0.7	0.9	

Table 1: The OpenBSD version in which vulnerabilities were introduced into the source code (born) and the version in which they were repaired (died). The final row, at the very bottom of the table, shows the count in millions of lines of code altered/introduced in that version.

1. We compiled a database of vulnerabilities identified in the 7.5 years between 19 May 1998 and 17 November 2005 by merging data from the OpenBSD security web page and four public vulnerability databases: NVD (formerly ICAT), Bugtraq, OSVDB, and ISS X-Force.

2. We examined CVS records and the source code to identify the date on which the vulnerability was repaired. If the fix was itself faulty, the date of the first repair effort is used because it most closely tracks the date of discovery. We then selected the earliest of two possible dates for the vulnerability's *death*: the date on which the vulnerability was reported or the date of the first repair.[2]

3. We manually examined prior versions of the source code to identify the date on which the vulnerability was introduced. If there was any doubt, the earliest possible date was chosen. A vulnerability is *born* on the date that the first version of OpenBSD to include the vulnerable code is released.

Not all vulnerabilities could be easily and precisely categorized: the process was manual, time-consuming, and laborious. In particular, we had to make a number of decisions about inclusion and uniqueness.

We included vulnerabilities that we believed to be applicable to the bulk of OpenBSD's installed base. We excluded vulnerabilities that were specific to processor architectures other than the x86. We also excluded vulnerabilities that were location/country dependent. In addition, we excluded reports of vulnerabilities in historical versions of OpenBSD if the release that was current at the time of the report was not vulnerable.

Our analysis covers all portions of the OpenBSD code in the primary CVS repository. This includes the X-windowing system, the Apache web server, and many additional services not traditionally considered to be part of the core operating system. However, this repository excludes the 'ports' collection of third-party software that is not officially part of OpenBSD. We included vulnerabilities regardless of whether or not they applied to the default configuration of OpenBSD.

Some of the reports in these vulnerability databases do not fit the traditional definition of a vulnerability: a few might be better described as reports of proactive efforts to improve security design. However, we did not exclude any vulnerability reports based on justification or severity, as we lacked an unbiased methodology with which to assess these factors.

The most difficult distinction for us to make was whether a group of related reports should be treated as independent vulnerabilities or a single vulnerability. Individuals may find and report multiple related vulnerabilities at once: either by discovering a new class of vulnerability, a new mechanism for identifying vulnerabilities, or a section of poorly written code. Often these related vulnerabilities are remediated in the same patch. In order to maintain the independence of each data point, we grouped closely-related vulnerabilities that were identified within a few days of each other into a single vulnerability data point. A discussion of the need for independent data points—and a more detailed explanation of how vulnerabilities were characterized in this data set—is described in earlier work [10].

Similarly, OpenBSD includes some software that is maintained by third parties (*e.g.* sendmail). Those third parties often release new versions of their software that bundle together fixes for multiple (previously secret) security flaws. Unfortunately, the third party producers do not always make available the information necessary to identify the birth and death date of the component vulnerabilities. As a result, every such 'bundle' patch was counted as a single vulnerability and was assigned the birth date of the youngest identifiable security flaw included in the bundle. Our decision to bundle vulnerabilities is a result of our inability to obtain access to the data necessary to differentiate between them. However, it may result in an inflated perception of security for the system: the models will process fewer vulnerabilities and thus may find a more rapid trend towards depletion.

3.2 Vulnerability births and deaths

Table 1 shows the number of vulnerabilities that were born and died in each version of OpenBSD. The version in which the vulnerability was born is specified by the column. The version in which the vulnerability died is specified by the row. The first column contains a total of 87 vulnerabilities that are foundational: they were introduced before the start of our study and were thus present in the code of the foundation version, 2.3. The top entry in that column indicates that 5 vulnerabilities died during the six months between the release of version 2.3 and the release of 2.4.

The bottom row of Table 1 also shows the number of lines of code, in millions, that were altered/introduced in each release (see Section 3.3 for the methodology used to obtain this information).

3.3 Source code composition

We analyzed the collective changes to the OpenBSD code repository in order to establish how much code was altered/introduced in each version.

We first pre-processed each version of the source code. Only files with the suffix .c or .h were retained, and all comments were stripped. Furthermore, files whose name included keywords indicating that they belonged to an architecture other than x86 were removed.

After pre-processing was completed, each version was compared with each successive version. We used `diff` to compare files with the same path and filename. The `diff` tool was instructed to ignore changes in whitespace or the location of line breaks.

The OpenBSD development team sometimes moved or copied files, which is difficult to track via CVS. To detect copies and moves, files with the same name but different paths were also compared. If they were found to be identical, we replicated the file in the earlier version at the directory in which it was found in the later version. (These replicas were used only to determine if code in future versions derived from earlier versions: they were not used to calculate the total line count.)

The estimate of code commonality is highly conservative. The `diff` tool marked code lines as changed even for trivial alterations like global variable renaming and some types of reformatting—and the OpenBSD team has been reformatting the code base. In addition, this process will indicate that all of the code in a file is new if that file was moved/copied and then had just one line altered between versions. (Recall that the automated comparison process only understands that a file was moved if the file in the new location is an exact copy of the file in the old location.) Furthermore, if the name of a file is changed then all of the code in that file is considered to be new. The comparison data will thus understate the degree to which later releases are composed of substantively unchanged code from earlier releases.

4 Analysis

We now address our first four questions about the security of OpenBSD, using the vulnerability and source code composition data sets described above.

4.1 How much does legacy code influence security today?

The majority (87 of 140, or 62%) of the vulnerabilities found during the period of the study are foundational; that is, they were born prior to the release of the foundation version. We considered two hypotheses to explain why reported vulnerabilities were so often foundational: foundational code might be of lower quality than more recent code, or foundational code may constitute the bulk of the total code base.

The source code history data supports the latter hypothesis. Even after 7.5 years and 14 newer versions, the foundation version dominates the overall source code: at least 61% of the lines of code in version 3.7 are foundational, unchanged since the release of version 2.3. As a

Source version \ Composite version	2.3	2.4	2.5	2.6	2.7	2.8	2.9	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
3.7															4
3.6														3	3
3.5													7	6	6
3.4												4	4	3	3
3.3											2	2	1	1	1
3.2										2	1	2	1	1	1
3.1									5	5	5	4	3	3	1
3.0								4	3	3	3	2	2	2	2
2.9							14	13	12	11	11	10	8	8	7
2.8						3	2	3	2	2	2	2	2	1	1
2.7					6	6	4	4	3	3	3	2	2	2	2
2.6				9	9	9	7	6	6	6	6	5	5	5	5
2.5			3	2	2	2	1	1	1	1	1	1	1	1	1
2.4		4	4	3	3	3	2	2	2	2	2	1	1	1	1
2.3	100	96	94	86	79	77	70	67	66	66	65	64	62	61	61

Table 2: The percentage of each version of OpenBSD that is composed from earlier versions. Columns represent *composite* versions of OpenBSD, whereas rows represent the *source* versions of OpenBSD from which they are composed. Each value in the table is the percentage of lines in the composite version that were last modified in the source version.

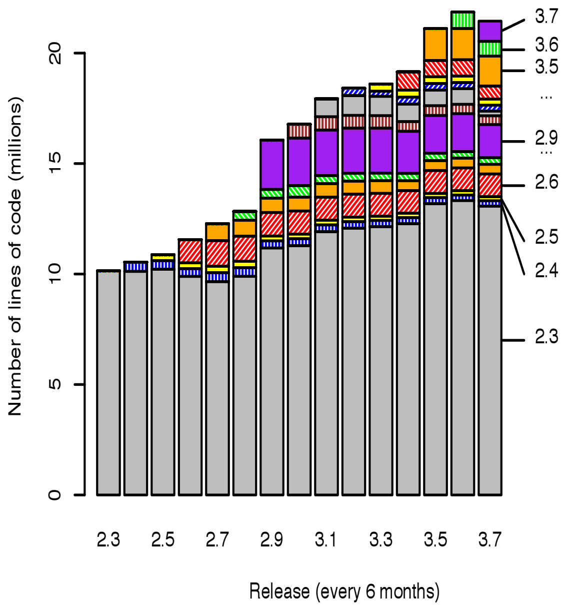


Figure 1: The composition of the full source code. The composition of each version is broken-down into the lines of code originating from that version and from each prior version.

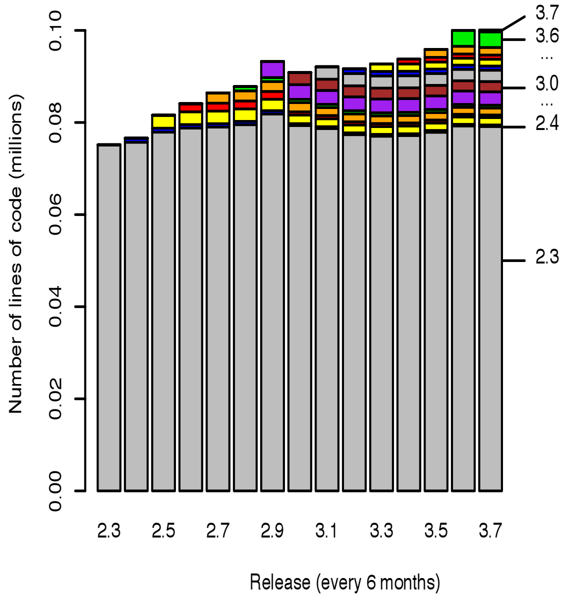


Figure 2: The composition of the source code directory within the kernel (sys/kern) that had the most vulnerabilities. The composition of each version is broken-down into the lines of code originating from that version and from each prior version.

result, the security of the foundation version may still be driving the overall security of OpenBSD.

Table 2 illustrates the proportion of each version of OpenBSD that is derived from earlier versions. Each column represents a composite version; each row represents a source version that contributes code to the composite. Values represent the percentage of the lines of code in the composite version that originate in the source version.[3] A line of code in a composite version of OpenBSD is said to originate in a source version if the line was last modified in that source version.

For example, the fifth column breaks down the composition of OpenBSD version 2.7. The top row of the column indicates that 6% of the lines of code originate in that version: they were either altered since the prior version or have been newly introduced. The second row from the top shows that 9% of the source code was altered/introduced in the prior version, 2.6, and was not changed after that version. The bottom row indicates that the bulk of the code in version 2.7 (79%) was both present in and remains unchanged since the foundation version.

Figure 1 shows a graphical representation of the composition of each version, using lines of code rather than percentages. Version 2.3 is composed of a single bar: by definition, all code in this foundation version is said to originate in it. For each successive version, a new bar is added to represent the lines of code that were altered/introduced in that release.

When we look more closely at Figure 1, several large alterations/introductions of code stand out: in versions 2.6, 2.9, and 3.5. The magnitude of the changes in versions 2.6 and 3.5 is primarily due to a large number of files being renamed and slightly altered. Our current methodology thus overstates the number of new lines of code and understates the contribution of code derived from earlier versions. The changes in version 2.9 are caused in part by the renaming of files; however, they were also the result of a major upgrade of the XFree86 package.

We were initially surprised that the number of lines of foundational code fluctuates both downwards and *upwards*. However, increases in the number of lines of foundational code are readily explained: source files unaltered since the foundation version were copied and used in other areas of the code.

Of all the second-level source code directories, the sys/kern directory contains the largest number of reported vulnerabilities. Fifteen of the seventeen vulnerabilities reported in this portion of the kernel were introduced in the foundation version. Figure 2 shows the evolving composition of the source code in the sys/kern directory. Many of the vulnerabilities in this subsystem have been in code related to the processing of signals: although this subsystem is part of the kernel, it does not include networking, file system, or virtual memory code. The code in one of the networking portions of the kernel (sys/netinet) has contributed ten vulnerabilities during the course of the study, seven of which are foundational.

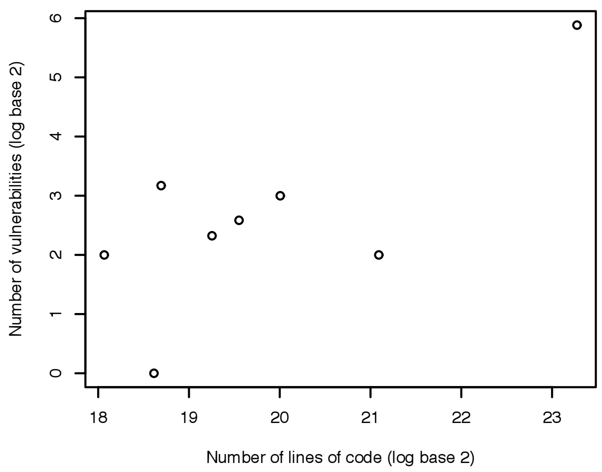


Figure 3: The number of vulnerabilities introduced and reported within four years of release compared to the number of lines of code altered/introduced, by version.

4.2 Do larger code changes have more vulnerabilities?

Software engineers have examined the defect density of code: the ratio of the number of defects in a program to the number of lines of code. Some have argued that any well-written code can be expected to have a defect density that falls within a certain range, *e.g.* 3–6 defects per thousand lines of code (KLOC) [5]. We thus ask whether or not there is a linear relationship between the number of lines of code altered/introduced in a version of OpenBSD and number of vulnerabilities introduced in that version.

As we cannot measure the total number of vulnerabilities present, we measure the number discovered within four years of release for each version that is at least four years old. The number of vulnerabilities reported during this period is reported in the third column of Table 3. The fourth column contains the *vulnerability density*: the ratio of vulnerabilities reported to the number of lines of code in that version. In this instance, densities are reported in units of vulnerabilities per millions of lines of code (MLOC).

Figure 3 illustrates the relationship between the number of lines of altered/introduced code and the number of vulnerabilities reported. The standard correlation test (Pearson's ρ) is not applicable because we do not have enough data points. A non-parametric correlation test,

		Reported within 4 yrs of release		*Reported by end of study*	
Vers.	MLOC	Vulns	$\frac{\text{Vulns}}{\text{MLOC}}$	Vulns	$\frac{\text{Vulns}}{\text{MLOC}}$
2.3	10.14	59	5.8	87	8.6
2.4	.42	9	21.4	14	33.0
2.5	.28	4	14.3	6	21.8
2.6	1.05	8	7.6	9	8.6
2.7	.77	6	7.8	7	9.1
2.8	.40	0	0.0	0	0.0
2.9	2.23	4	1.8	4	1.8
3.0	.63	5	7.9	5	8.0
3.1	.81			2	2.5
3.2	.33			2	6.0
3.3	.32			2	6.2
3.4	.83			0	0.0
3.5	1.44			2	1.4
3.6	.74			0	0.0
3.7	.91			0	0.0
Total	21.30	95	6.0	140	6.6

Table 3: Vulnerability and code modification statistics for each version of OpenBSD. MLOC is the number of lines of code (in millions) altered/introduced in each version.

Spearman's ρ, is unable reject the null hypothesis that there is no correlation: it calculates a correlation coefficient of 0.53 and a p-value of 0.18.[4]

We are thus unable to find a significant correlation between the number of lines of altered/introduced code and the number of vulnerabilities reported.

4.3 Do today's coders introduce fewer vulnerabilities per line of code?

The vulnerability density of code added in new OpenBSD releases could provide an indication of the success of their efforts to produce secure code. On the other hand, code added by the OpenBSD team often provides security functionality: *e.g.* OpenSSH. As a result, that code is likely to attract a disproportionate share of attention from individuals searching for vulnerabilities; this extra attention may account for any differences between the versions' vulnerability densities.

For each release, Table 3 shows the number of reported vulnerabilities, the number of lines of code altered/introduced (in millions), and the vulnerability density. The third column shows the number of vulnerabilities reported within four years of each version's release, and the fourth column shows the corresponding vulnerability density. The fifth column shows the number of vulnerabilities reported during the entire study, and the sixth column shows the corresponding vulnerability density.

The vulnerability density of the foundation version is in the middle of the pack. Versions 2.4 and 2.5 stand out for having the highest vulnerability densities (33 and 21.8 reported per million lines of code at the end of the study, respectively).

The large ratio of reported vulnerabilities per line of code in version 2.4 seems to support the intuition that code providing security functionality is more likely to contain vulnerabilities. Version 2.4 saw the introduction of the Internet Key Exchange (IKE) key management daemon (isakmpd, two vulnerabilities introduced) and OpenSSL (three vulnerabilities introduced). As a result, the new code added in that release may have drawn particular attention from vulnerability hunters.

In version 2.5, two of the six vulnerabilities introduced were in the Apache program.

The density of reported vulnerabilities for code originating in versions 2.6, 2.9, and 3.5 are lower in part because of the inflated new-code counts for those versions (see Section 3.3).

When calculated per *thousand* lines of code, rather than per million, the density of *all* reported vulnerabilities ranged from 0–0.033 and averaged 0.00657. As noted above, some software engineers estimate the defect density of well-written code to be 3–6 per thousand lines of code [5]; these vulnerability densities are three orders of magnitude less than that amount. The two figures are not necessarily contradictory: defects include both vulnerabilities and bugs that are not vulnerabilities.

4.4 What is the median lifetime of a vulnerability?

Rescorla [11] applies an exponential model to his data, so he is able to ascertain the half-life of the vulnerability sets he considers: those half-lives range from 10 months to 3.5 years. Unfortunately, exponential models do not fit our data set (see Section 5). As a result, we are not able to ascertain, in a formal sense, the half-life of vulnerabilities in OpenBSD. Instead, we calculate the median lifetime of reported vulnerabilities: the time elapsed between the release of a version and the death of half of the vulnerabilities reported in that version.

Figure 4 plots the age, at report, of foundational vulnerabilities. The data is necessarily right censored: we do not know that we have found all of the vulnerabilities in the foundation version. This data thus provides a lower-bound of 2.6 years (961 days) on the median lifetime of foundational vulnerabilities.

Is the median lifetime of vulnerabilities decreasing in newer versions? Table 4 depicts this time for those vulnerabilities identified within six years of the release

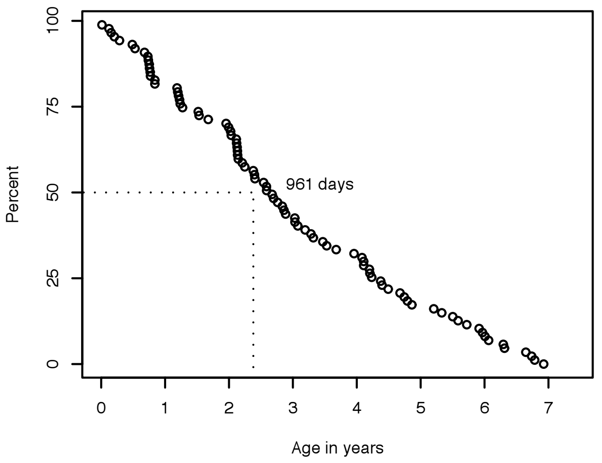


Figure 4: The lifetime of foundational vulnerabilities reported during the study period.

Version	Median lifetime
2.3	878
2.4	1288
2.5	445
2.6	645

Table 4: The median lifetime of vulnerabilities reported within the first six years of a version's release.

of versions 2.3, 2.4, 2.5, and 2.6; this data relies upon the gross simplifying assumption that all vulnerabilities present were found within six years of each version's release. (We make this assumption so that we include the same time span after release for each version.) The results do not indicate a trend. During the course of the study, six vulnerabilities were identified that had been introduced in version 2.5: only five of those fell within the first six years after it's release. This lack of data partially explains the low median lifetime of vulnerabilities for version 2.5, and it highlights the limitations of this analysis.

The most striking part of this analysis is that the median lifetime of vulnerabilities is so long.

5 Are reporting rates declining?

We now address whether or not there has been a decline in the rate at which foundational vulnerabilities have been reported.

5.1 Illustrating reporting trends

Figures 5 and 6 categorize foundational vulnerabilities by the time period in which they were reported: we divide the study into periods of equal length.

The columns in Figure 5 represent the number of vulnerabilities reported in each of eight periods. The confidence intervals are derived from a normal approximation of a homogenous Poisson process. The confidence intervals are too large to permit effective analysis: by visual inspection alone, one can see that an exponential, S-shaped, or even a linear model could fit within these bounds.

However, more conclusive results can be obtained by dividing the study period into halves, as shown in Figure 6. The number of vulnerabilities reported significantly declines from the first period (58 vulnerabilities) to the second (28 vulnerabilities). The 95% confidence interval for the first period ranges from 43.1 to 72.9; for the second period, it ranges from 17.6 to 38.4.

Another way to examine the frequency of vulnerability reports is to measure the time between them. An analogous metric from reliability engineering, the *time-between-failures* (TBF), can be applied by defining a failure as the report of a vulnerability. Figure 7 groups foundational vulnerability reports by their time-between-failures. Each group appears as a pair of columns. The dark gray columns, the first column in each pair, represent vulnerabilities reported during the first half of the study. The light gray columns, the second column in each pair, represent vulnerabilities reported in the last half of the study.

Figure 7 shows that the second half of the study had far fewer foundational vulnerabilities with TBFs of 25 or less than the first half of the study (39 in the first half vs. 13 in the second half); the number of vulnerabilities with TBFs greater than 25 did not significantly change between the two halves (17 in the first half vs. 18 in the second half). The TBF ranges were chosen by dividing by five the maximum TBF of 126.

5.2 Analyzing reporting trends

We find a downward trend in the rate of vulnerability discovery, a result which contradicts previous work by Eric Rescorla [11]. His analysis failed to reject the hypothesis that the rate of vulnerability reporting has remained constant in three of the four operating systems

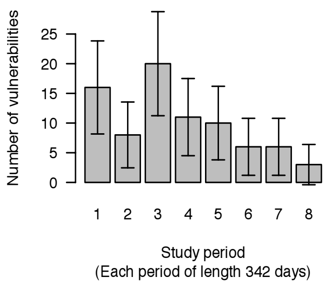


Figure 5: The number of foundational vulnerabilities reported during each eighth of the study.

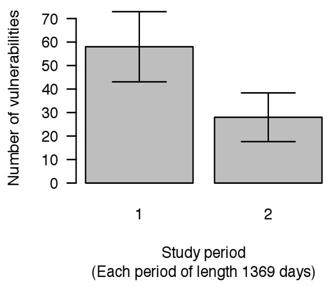


Figure 6: The number of foundational vulnerabilities reported during each half of the study.

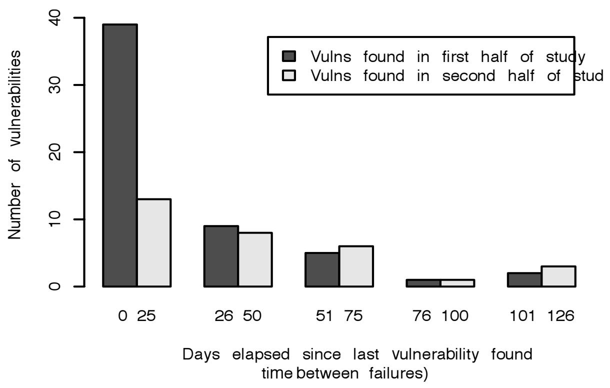


Figure 7: The number of days between reports of foundational vulnerabilities reported in the first half of the study compared with those reported in the second half.

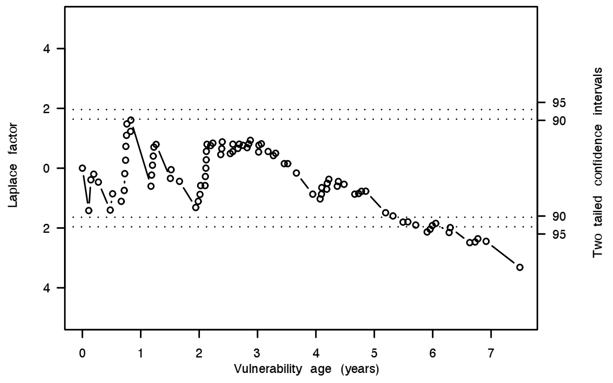


Figure 8: Laplace test for the existence and direction of a trend in the rate of vulnerability reporting.

Time-between-failures data	Number of days
Mean	29.1
Median	18
σ	29.14
Minimum	1
Maximum	126

Initial intensity	0.051
Current intensity	0.024
Purification level	0.676

Table 5: Measurements & predictions for foundational vulnerabilities.

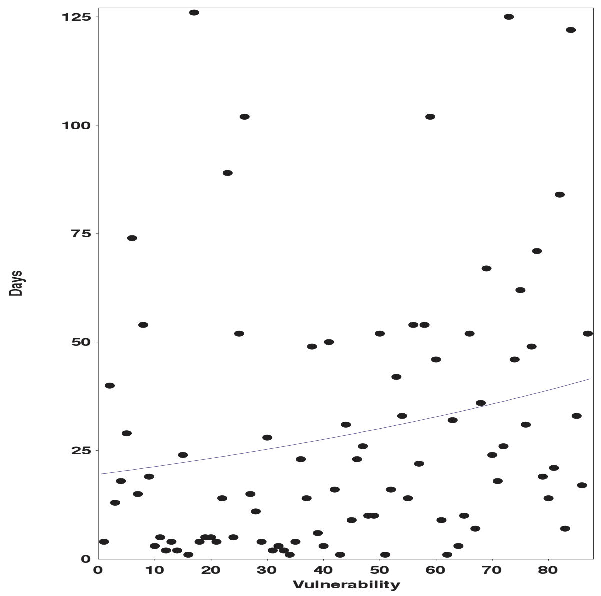


Figure 9: Fitted Musa's Logarithmic model for foundational vulnerability report intervals. The vertical axis shows the time-between-failures: the number of days that have passed since the prior vulnerability was found.

he evaluated.

Our analysis above indicates a clear decrease in the rate of reporting of foundational vulnerabilities. In addition, we applied a Laplace test to make our data more directly comparable to the work of Rescorla. In the Laplace test, the discovery of vulnerabilities is assumed to be a Poisson process; the test assesses whether there is a decreasing or increasing trend with respect to inter-arrival times. The data we used were the number of days elapsed between the identification of each successive foundational vulnerability. These data are equivalent to those for time-between-failures in traditional models of reliability.

The lowest horizontal dotted line in Figure 8 is at a Laplace factor of -1.96. When the calculated Laplace factors are less than that amount, the data indicates a decreasing rate of vulnerability reporting with a two-tailed confidence level of 95%. The test finds evidence for a decrease in the rate of vulnerability reporting by the end of year four; by the end of year six, the evidence for a decrease in the reporting rate is statistically significant.

This test therefore supports our conclusion that the rate at which foundational vulnerabilities are reported is declining.

5.3 Fitting vulnerability data to reliability growth models

The case that OpenBSD is becoming more secure can also be supported using reliability growth models. While normally applied to the more random discovery of defects, these models can also be applied to the reporting of vulnerabilities. Rescorla also applied two reliability growth models to his larger, less precise, data set. His results for both models matched his results with the Laplace test: he could not fit the models to three of the four operating systems he evaluated.

We analyzed the data with seven time-between-failures reliability growth models. One of the seven models had acceptable one-step-ahead predictive accuracy and goodness-of-fit for the data set: Musa's Logarithmic model.[5]

The estimates produced by Musa's Logarithmic model are presented in Table 5. The intensity is the number of vulnerabilities expected to be reported on a given day. The intensity on the first day of the study (the *initial intensity*) is 0.051; by the end of the study, the intensity has been more than halved, to 0.024 (the *current intensity*).

The *purification level* is a normalized estimate of how vulnerability-free the program is at the end of the period covered by the data set. A purification level of one would indicate a program entirely free of vulnerabilities [15]. Musa's Logarithmic model calculates that 67.6% of all foundational vulnerabilities were reported during the study.

Figure 9 shows the successfully fitted Musa's Logarithmic model superimposed over the data set. The y-axis indicates the time-between-failures in days (the number of days that elapsed since the prior vulnerability was reported). Although the data points are widely scattered, the model indicates a trend toward an increasing time between vulnerability reports. In particular, far fewer of the later vulnerabilities are reported within ten days of each other.

The reliability growth analysis thus indicates that the rate of foundational vulnerabilities reported is decreasing.

6 Related Work

Our study builds on prior work in software reliability growth and on efforts to characterize the social utility of finding and reporting vulnerabilities.

We have applied models that define software reliability in terms of the number of faults in a body of code. Such models "apply statistical techniques to the observed failures during software testing and operation to forecast the product's reliability" [2, p. 6]. As faults are identified and removed, the system will fail less frequently and hence be more reliable. These models can thus be utilized to estimate characteristics about the number of faults remaining in the system and when those faults may cause failures. These estimates can be then be used to gauge the amount of further testing required to meet reliability requirements.

Eric Rescorla first applied reliability growth models to post-release vulnerability reporting data in order to question the social utility of publicly disclosing vulnerabilities [11]. He found no clear trend reduction in the rate of vulnerability reporting, and he estimates that the half-life of a vulnerability is between 10 months and 3.5 years. However, the ICAT database he uses is not focused on vulnerability age, and it does not reliably report the dates on which vulnerabilities were born.[6] For our analysis, we used the version control system to ascertain the exact date of birth for each vulnerability in our data set. Furthermore, we test more models (seven) than the two that he evaluated. In addition, we only present results from the model that passed both goodness-of-fit and one-step-ahead predictive accuracy tests; Rescorla only utilized the former test.

Another related body of literature looks at measuring software security through market-mechanisms. L. Jean Camp and Catherine Wolfram proposed a market through which vulnerability credits could be traded; such markets have worked previously to create incentives for the reduction of negative externalities like environmental pollutants [3].

Prior to this collaboration, Stuart Schechter proposed creating markets for reports of previously undiscovered vulnerabilities, in order to measure software security. He argued that the bid, ask, and most recent sale prices in such a market approximate the labor cost to find a vulnerability. He further argued that these prices can establish which of two products the market deems to have vulnerabilities that are less expensive to find [12], [13]. Andy Ozment has separately proposed that a vulnerability market could be better designed as an auction; he then used the economic literature on auctions to refine the proposed design [9].

Given the emergence of a black market for reports of undiscovered vulnerabilities, metrics that estimate the cost to discover a vulnerability may be more valuable than those that measure the reporting rate. Several organizations are now actively purchasing vulnerabilities, so an open market or auction as described in this literature is not infeasible. Unfortunately, the business models of some of these organizations are not socially optimal [6]. Furthermore, these organizations are not sharing pricing information, hindering the movement toward an open market or auction. Until such an entity or entities arise—and until that entity has gathered several years of data—other means of measuring software security are necessary.

One path forward for future research into vulnerability reporting rates is to employ more sophisticated modeling techniques. The reliability growth literature is rich with means of improving models' accuracy. In addition, vulnerability analysis can be combined with traditional 'software metrics:' metrics that attempt to measure a program's size, complexity, *etc*. If performed with an awareness of previous failures in this field, this line of research might lead to other fruitful measurements of or predictors of the rate of vulnerability discovery.

In future work, we plan to examine a competing operating system and compare the rate of vulnerability reporting in that product with the rate in OpenBSD. We hope to provide further insight on the success of secure coding measures, the level of effort expended to find vulnerabilities, and changes to the rate of vulnerability reporting in newly introduced code.

7 Conclusion

Over a period of 7.5 years and fifteen releases, 62% of the 140 vulnerabilities reported in OpenBSD were *foundational*: present in the code at the beginning of the study. It took more than two and a half years for the first half of these foundational vulnerabilities to be reported.

We found that 61% of the source code in the final version studied is foundational: it remains unaltered from the initial version released 7.5 years earlier. The rate of reporting of foundational vulnerabilities in OpenBSD is thus likely to continue to greatly influence the overall rate of vulnerability reporting.

We also found statistically significant evidence that the rate of foundational vulnerability reports decreased during the study period. We utilized a reliability growth model to estimate that 67.6% of the vulnerabilities in the foundation version had been found. The model's estimate of the expected number of foundational vulnerabilities reported per day decreased from 0.051 at the start of the study to 0.024.

Acknowledgements

The authors thank Rich Lippman for his advice on the statistical analysis. We also greatly appreciate the comments and feedback provided by Ross Anderson, Robert Cunningham, David Purdy, and Eric Rescorla.

Notes

[1]ICAT is now known as the National Vulnerability Database (NVD) [7].

[2]The release of a public report and the repair of the vulnerability do not always occur in the same order. When a vulnerability is reported to an entity other than the OpenBSD development team, the date of the public report often precedes the date on which a repair is committed to CVS. When a vulnerability is reported directly to the OpenBSD development team, they usually commit a repair into the CVS repository prior to publicly announcing the vulnerability. We utilize the earlier of the two dates so that we most closely approximate the date of actual discovery.

[3]Because the percentages were rounded, the total percentage for each version may not exactly equal one hundred.

[4]A correlation coefficient of 1 would indicate a positive linear correlation, -1 would indicate a negative linear correlation, and 0 indicates no correlation.

[5]The SMERFS^3 reliability growth modeling tool was used to assess the models [14]. Musa's Logarithmic model had acceptable bias (0.13), noise (0.40), trend (0.09), and Kolmogorov distance goodness-of-fit (0.09397) results. Bias is determined by a μ-plot; it assesses the absolute predictive accuracy of the models. The noise and trend results are useful primarily to ensure that the predictive accuracy indicated by the μ-plot results was not due to opposing trends of inaccuracy canceling each other out on the average. For a more detailed explanation of the acceptability tests, see [1].

[6]In particular, the ICAT database may omit the fact that out-of-date versions of a program include a vulnerability. As a result, vulnerabilities may appear to have been introduced in much newer versions of a program than is actually the case.

References

[1] Abdel-Ghaly, A. A., Chan, P. Y., and Littlewood, B. Evaluation of competing software reliability predictions. *IEEE Transactions on Software Engineering 12*, 9 (1986), 950–967.

[2] AIAA/ANSI. *Recommended Practice: Software Reliability*. ANSI, 1993. R-013-1992.

[3] Camp, L., and Wolfram, C. Pricing security. In *Proceedings of the CERT Information Survivability Workshop* (Oct. 2000), pp. 31–39. Boston, MA, USA.

[4] de Raadt, T. Exploit mitigation techniques (in OpenBSD, of course). In *Proceedings of OpenCON 2005* (Nov. 2005). Venice, Italy.

[5] Hatton, L. Re-examining the fault density - component size connection. *IEEE Software 14*, 2 (1997), 89–97.

[6] Kannan, K., and Telang, R. Economic analysis of market for software vulnerabilities. In *Workshop on Economics and Information Security* (May 2004). Minneapolis, MN, USA.

[7] NIST. NVD metabase: A CVE based vulnerability database. `http://nvd.nist.gov`.

[8] OpenBSD. CVS – OpenBSD security page, revision 1.12, Feb. 1998. `http://www.openbsd.org/cgi-bin/cvsweb/~checkout~/www/security.html?rev=1.12&content-type=text/html`.

[9] Ozment, A. Bug auctions: Vulnerability markets reconsidered. In *Workshop on Economics and Information Security* (May 2004). Minneapolis, MN, USA.

[10] Ozment, A. Software security growth modeling: Examining vulnerabilities with reliability growth models. In *Proceedings of the First Workshop on Quality of Protection* (September 2005). Milan, Italy.

[11] Rescorla, E. Is finding security holes a good idea? In *Workshop on Economics and Information Security* (May 2004). Minneapolis, Minnesota.

[12] Schechter, S. How to buy better testing: Using competition to get the most security and robustness for your dollar. In *Infrastructure Security Conference* (Oct. 2002). Bristol, UK.

[13] Schechter, S. Quantitatively differentiating system security. In *Workshop on Economics and Information Security* (May 2002). Berkeley, CA, USA.

[14] Stoneburner, W. SMERFS (Statistical Modeling and Estimation of Reliability Functions for Systems), Jan. 2003. `http://www.slingcode.com/smerfs/`.

[15] Tian, J. Integrating time domain and input domain analyses of software reliability using tree-based models. *IEEE Transactions on Software Engineering 21*, 12 (Dec. 1995), 945–958.

N-Variant Systems
A Secretless Framework for Security through Diversity

Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill,
Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser
University of Virginia, Department of Computer Science
http://www.nvariant.org

Abstract

We present an architectural framework for systematically using automated diversity to provide high assurance detection and disruption for large classes of attacks. The framework executes a set of automatically diversified variants on the same inputs, and monitors their behavior to detect divergences. The benefit of this approach is that it requires an attacker to simultaneously compromise all system variants with the same input. By constructing variants with disjoint exploitation sets, we can make it impossible to carry out large classes of important attacks. In contrast to previous approaches that use automated diversity for security, our approach does not rely on keeping any secrets. In this paper, we introduce the N-variant systems framework, present a model for analyzing security properties of N-variant systems, define variations that can be used to detect attacks that involve referencing absolute memory addresses and executing injected code, and describe and present performance results from a prototype implementation.

1. Introduction

Many security researchers have noted that the current computing monoculture leaves our infrastructure vulnerable to a massive, rapid attack [70, 29, 59]. One mitigation strategy that has been proposed is to increase software diversity. By making systems appear different to attackers, diversity makes it more difficult to construct exploits and limits an attack's ability to propagate. Several techniques for automatically producing diversity have been developed including rearranging memory [8, 26, 25, 69] and randomizing the instruction set [6, 35]. All these techniques depend on keeping certain properties of the running execution secret from the attacker. Typically, these properties are determined by a secret key used to control the randomization. If the secret used to produce a given variant is compromised, an attack can be constructed that successfully attacks that variant. Pointer obfuscation techniques, memory address space randomization, and instruction set randomization have all been demonstrated to be vulnerable to remote attacks [55, 58, 64]. Further, the diversification secret may be compromised through side channels, insufficient entropy, or insider attacks.

Our work uses artificial diversity in a new way that does not depend on keeping secrets: instead of diversifying individual systems, we construct a single system containing multiple variants designed to have disjoint exploitation sets. Figure 1 illustrates our framework. We refer to the entire server as an N-variant system. The system shown is a 2-variant system, but our framework generalizes to any number of variants. The polygrapher takes input from the client and copies it to all the variants. The original server process P is replaced with the two variants, P_0 and P_1. The variants maintain the client-observable behavior of P on all normal inputs. They are, however, artificially diversified in a way that makes them behave differently on abnormal inputs that correspond to an attack of a certain class. The monitor observes the behavior of the variants to detect divergences which reveal attacks. When a divergence is detected, the monitor restarts the variants in known uncompromised states.

As a simple example, suppose P_0 and P_1 use disjoint memory spaces such that any absolute memory address that is valid in P_0 is invalid in P_1, and vice versa. Since the variants are transformed to provide the same semantics regardless of the memory space used, the behavior

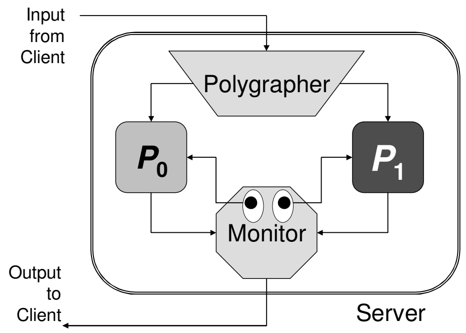


Figure 1. N-Variant System Framework.

on all normal inputs is identical (assuming deterministic behavior, which we address in Section 5). However, if an exploit uses an absolute memory address directly, it must be an invalid address on one of the two variants. The monitor can easily detect the illegal memory access on the other variant since it is detected automatically by the operating system. When monitoring is done at the system call level, as in our prototype implementation, the attack is detected before any external state is modified or output is returned to the attacker.

The key insight behind our approach is that in order for an attacker to exploit a vulnerability in *P*, a pathway must exist on one of the variants that exploits the vulnerability without producing detectably anomalous behavior on any of the other variants. If no such pathway exists, there is no way for the attacker to construct a successful attack, even if the attacker has complete knowledge of the variants. Removing the need to keep secrets means we do not need to be concerned with probing or guessing attacks, or even with attacks that take advantage of insider information.

Our key contributions are:

1. Introducing the N-variant systems framework that uses automated diversity techniques to provide high assurance security properties without needing to keep any secrets.
2. Developing a model for reasoning about N-variant systems including the definition of the normal equivalence and detection properties used to prove security properties of an ideal N-variant system (Section 3).
3. Identifying two example techniques for providing variation in N-variant systems: the memory address partitioning technique (introduced above) that detects attacks that involve absolute memory references and the instruction tagging technique that detects attempts to execute injected code (Section 4).
4. Describing a Linux kernel system implementation and analyzing its performance (Section 5).

In this paper we do not address recovery but consider it to be a successful outcome when our system transforms an attack that could compromise privacy and integrity into an attack that at worst causes a service shutdown that denies service to legitimate users. It has not escaped our attention, however, that examining differences between the states of the two variants at the point when an attack is detected provides some intriguing recovery possibilities. Section 6 speculates on these opportunities and other possible extensions to our work.

2. Related Work

There has been extensive work done on eliminating security vulnerabilities and mitigating attacks. Here, we briefly describe previous work on other types of defenses and automated diversity, and summarize related work on redundant processing and design diversity frameworks.

Other defenses. Many of the specific vulnerabilities we address have well known elimination, mitigation and disruption techniques. Buffer overflows have been widely studied and numerous defenses have been developed including static analysis to detect and eliminate the vulnerabilities [66, 67, 39, 23], program transformation and dynamic detection techniques [19, 5, 30, 45, 49, 57] and hardware modifications [38, 40, 41, 64]. There have also been several defenses proposed for string format vulnerabilities [56, 20, 63, 47]. Some of these techniques can mitigate specific classes of vulnerabilities with less expense and performance overhead than is required for our approach. Specific defenses, however, only prevent a limited class of specific vulnerabilities. Our approach is more general; it can mitigate all attacks that depend on particular functionality such as injecting code or accessing absolute addresses.

More general defenses have been proposed for some attack classes. For example, no execute pages (as provided by OpenBSD's W^X and Windows XP Service Pack 2) prevent many code injection attacks [2], dynamic taint analysis tracks information flow to identify memory corruption attacks [43], and control-flow integrity can detect attacks that corrupt an application to follow invalid execution paths [1]. Although these are promising approaches, they are limited to particular attack classes. Our framework is more general in the sense that we can construct defense against any attacker capability that can be varied across variants in an N-variant system.

Automated diversity. Automated diversity applies transformations to software to increase the difficulty an attacker will face in exploiting a security vulnerability in that software. Numerous transformation techniques have been proposed including rearranging memory [26, 8, 69, 25], randomizing system calls [17], and randomizing the instruction set [6, 35]. Our work is complementary to work on producing diversity; we can incorporate many different sources of variation as long as variants are constructed carefully to ensure the disjointedness required by our framework. A major advantage of the N-variant systems approach is that we do not rely on secrets for our security properties. This means we can employ diversification techniques with low entropy, so long as the transformations are able to produce variants with disjoint exploitation sets. Holland, Lim, and Seltzer propose many low entropy diversification techniques including number representations, register sets, stack direction, and memory layout [31]. In addition, our approach is not vulnerable to the type of secret-breaking attacks that have been demonstrated against secret-based diversity defenses [55, 58, 64].

O'Donnell and Sethu studied techniques for distributing diversity at the level of different software packages in a network to mitigate spreading attacks [44]. This can limit the ability of a worm exploiting a vulnerability present in only one of the software packages to spread on a network. Unlike our approach, however, even at the network level an attacker who discovers vulnerabilities in more than one of the software packages can exploit each of them independently.

Redundant execution. The idea of using redundant program executions for various purposes is not a new one. Architectures involving replicated processes have been proposed as a means to aid debugging, to provide fault tolerance, to improve dependability, and more recently, to harden vulnerable services against attacks.

The earliest work to consider running multiple variants of a process of which we are aware is Knowlton's 1968 paper [37] on a variant technique for detecting and localizing programming errors. It proposed simultaneously executing two programs which were logically equivalent but assembled differently by breaking the code into fragments, and then reordering the code fragments and data segments with appropriate jump instructions inserted between code fragments to preserve the original program semantics. The CPU could run in a checking mode that would execute both programs in parallel and verify that they execute semantically equivalent instructions. The variants they used did not provide any guarantees, but provided a high probability of detecting many programming errors such as out-of-range control transfers and wild memory fetches.

More recently, Berger and Zorn proposed a redundant execution framework with multiple replicas each with a different randomized layout of objects within the heap to provide probabilistic memory safety [7]. Since there is no guarantee that there will not be references at the same absolute locations, or reachable through the same relative offsets, their approach can provide only probabilistic expectations that a memory corruption will be detected by producing noticeably different behavior on the variants. Their goals were to enhance reliability and availability, rather than to detect and resist attacks. Consequently, when variations diverge in their framework, they allow the agreeing replicas to continue based on the assumption that the cause of the divergence in the other replicas was due a memory flaw rather than a successful attack. Their replication framework only handles processes whose I/O is through standard in/out, and only a limited number of system calls are caught in user space to ensure all replicas see the same values. Since monitoring is only on the standard output, a compromised replica could be successfully performing an attack and, as long as it does not fill up its standard out buffer, the monitor would not notice. The key difference between their approach and ours, is that their approach is probabilistic whereas our variants are constructed to guarantee disjointedness with respect to some property, and thereby can provide guarantees of invulnerability to particular attack classes. A possible extension to our work would consider variations providing probabilistic protection, such as the heap randomization technique they used, to deal with attack classes for which disjointedness is infeasible.

Redundant processing of the same instruction stream by multiple processors has been used as a way to provide fault-tolerance by Stratus [68] and Tandem [32] computers. For example, Integrity S2 used triple redundancy in hardware with three synchronized identical processors executing the same instructions [32]. A majority voter selects the majority output from the three processors, and a vote analyzer compares the outputs to activate a failure mode when a divergence is detected. This type of redundancy provides resilience to hardware faults, but no protection against malicious attacks that exploit vulnerabilities in the software, which is identical on all three processors. Slipstream processors are an interesting variation of this, where two redundant versions of the instruction stream execute, but instructions that are dynamically determined to be likely to be unnecessary are removed from the first stream which executes speculatively [60]. The second stream executes behind the first stream, and the processor detects inconsistencies between the two executions. These deviations either indicate false predications about unnecessary computations (such as a mispredicted branch) or hardware faults.

The distributed systems community has used active replication to achieve fault tolerance [9, 10, 16, 18, 50]. With active replication, all replicas are running the same software and process the same requests. Unlike our approach, however, active replication does nothing to hide design flaws in the software since all replicas are running the same software. To mitigate this problem, Schneider and Zhou have suggested proactive diversity, a technique for periodically randomizing replicas to justify the assumption that server replicas fail independently and to limit the window of vulnerability in which replicas are susceptible to the same exploit [51]. Active replication and N-variant systems are complementary approaches. Combining them can provide the benefits of both approaches with the overhead and costs associated with either approach independently.

Design diversity frameworks. The name *N-variant systems* is inspired by, but fundamentally different from, the technique known as *N-version programming* [3, 14]. The N-version programming method uses several independent development groups to develop different implementations of the same specification with the hope that different development groups will produce versions without common faults. The use of N-version programming to help with system security was proposed by Joseph [33]. He analyzed design diversity as manifest in N-version programming to see whether it could defeat certain attacks and developed an analogy between faults in computing systems that might affect reliability and vulnerabilities in computer systems that might affect security. He argued that N-version programming techniques might allow vulnerabilities to be masked. However, N-version programming provides no guarantee that the versions produced by different teams will not have common flaws. Indeed, experiments have shown that common flaws in implementations do occur [36]. In our work, program variants are created by mechanical transformations engineered specifically to differ in particular ways that enable attack detection. In addition, our variants are produced mechanically, so the cost of multiple development teams is avoided.

Three recent projects [46, 62, 28] have explored using design diversity in architectures similar to the one we propose here in which the outputs or behaviors of two diverse implementations of the same service (e.g., HTTP servers Apache on Linux and IIS on Windows) are compared and differences above a set threshold indicate a likely attack. The key difference between those projects and our work is that whereas they use diverse available implementations of the same service, we use techniques to artificially produce specific kinds of variation. The HACQIT project [34, 46] deployed two COTS web servers (IIS running on Windows and Apache running on Linux) in an architecture where a third computer forwarded all requests to both servers and compared their responses. A divergence was detected when the HTTP status code differed, hence divergences that caused the servers to modify external state differently or produce different output pages would not be detected. The system described by Totel, Majorczyk, and Mé extended this idea to compare the actual web page responses of the two servers [62]. Since different servers do not produce exactly the same output on all non-attack requests because of nondeterminism, design differences in the servers, and host-specific properties, they developed an algorithm that compares a set of server responses to determine which divergences are likely to correspond to attacks and which are benign. The system proposed by Gao, Reiter, and Song [28] deployed multiple servers in a similar way, but monitored their behavior using a distance metric that examined the sequence of system calls each server made to determine when the server behaviors diverged beyond a threshold amount.

All of these systems use multiple available implementations of the same service running on isolated machines and compare the output or aspects of the behavior to notice when the servers diverged. They differ in their system architectures and in how divergences are recognized. The primary advantage of our work over these approaches is the level of assurance automated diversity and monitoring can provide over design diversity. Because our system takes advantage of knowing exactly how the variants differ, we can make security claims about large attack classes. With design diversity, security claims depend on the implementations being sufficiently different to diverge noticeably on the attack (and functionality claims depend on the behaviors being sufficiently similar not exceed the divergence threshold on non-attack inputs). In addition, these approaches can be used only when diverse implementations of the same service are available. For HTTP servers, this is the case, but for custom servers the costs of producing a diverse implementation are prohibitive in most cases. Further, even though many HTTP servers exist, most advanced websites take advantages of server-specific functionality (such as server-side includes provided by Apache), so would not work on an alternate server. Design diversity approaches offer the advantage that they may be able to detect attacks that are at the level of application semantics rather than low-level memory corruption or code injection attacks that are better detected by artificial diversity. In Section 6, we consider possible extensions to our work that would combine both approaches to provide defenses against both types of attacks.

3. Model

Our goal is to show that for all attacks in a particular attack class, if one variant is compromised by a given attack, another variant must exhibit divergent behavior that is detected by the monitor. To show this, we develop a model of execution for an N-variant system and define two properties the variant processes must maintain to provide a detection guarantee.

We can view an execution as a possibly infinite sequence of states: $[S_0, S_1, \ldots]$. In an N-variant system, the state of the system can be represented using a tuple of the states of the variants (for simplicity, this argument assumes the polygrapher and monitor are stateless; in our implementation, they do maintain some state but we ignore that in this presentation). Hence, an execution of an N-variant system is a sequence of state-tuples where $S_{t,v}$ represents the state of variant v at step t: $[<S_{0,0}, S_{0,1}, \ldots S_{0,N-1}>, <S_{1,0}, S_{1,1}, \ldots S_{1,N-1}>, \ldots]$.

Because of the artificial variation, the concrete state of each variant differs. Each variant has a *canonicalization function*, C_v, that maps its state to a canonical state that matches the corresponding state for the original process. For example, if the variation alters memory addresses, the mapping function would need to map the variant's altered addresses to canonical addresses. Under normal execution, at every execution step the canonicalized states of all variants are identical to the original program state:

$$\forall t \geq 0, 0 \leq v < N, 0 \leq w < N:$$
$$C_v (S_{t,v}) = C_w (S_{t,w}) = S_t.$$

Each variant has a *transition function*, T_v, that takes a state and an input and produces the next state. The original program, P, also has a transition function, T. The set of possible transitions can be partitioned into *consistent transitions* and *aberrant transitions*. Consistent transitions take the system from one normal state to another normal state; aberrant transitions take the system from a normal state to a compromised state. An attack is successful if it produces an aberrant transition without detection. Our goal is to detect all aberrant transitions.

We partition possible variant states into three sets: *normal*, *compromised*, and *alarm*. A variant in a normal state is behaving as intended. A variant in a compromised state has been successfully compromised by a malicious attack. A variant in an alarm state is anomalous in a way that is detectable by the monitor. We aim to guarantee that the N-variant system never enters a state-tuple that contains one or more variants in comprised states without any variants in alarm states. To establish this we need two properties: *normal equivalence* and *detection*.

Normal equivalence. The normal equivalence property is satisfied if the N-variant system synchronizes the states of all variants. That is, whenever all variants are in normal states, they must be in states that correspond to the same canonical state. For security, it is sufficient to show the variants remain in equivalent states. For correctness, we would also like to know the canonical state of each of the variants is equivalent to the state of the original process.

We can prove the normal equivalence property statically using induction:

1. Show that initially all variants are in the same canonical state: $\forall\ 0 \leq v < N$: $C_i (S_{0,v}) = S_0$.
2. Show that every normal transition preserves the equivalence when the system is initially in a normal state:

$$\forall S \in Normal, 0 \leq v < N, S_v$$
$$\text{where } C_v (S_v) = S, p \in Inputs:$$
$$C_v (T_v (S_v, p)) = T (S, p).$$

Alternatively, we can establish it dynamically by examining the states of the variants and using the canonicalization function to check the variants are in equivalent states after every step. In practice, neither a full static proof nor a complete dynamic comparison is likely to be feasible for real systems. Instead, we argue that our implementation provides a limited form of normal equivalence using a combination of static argument and limited dynamic comparison, as we discuss in Section 5.

Detection. The detection property guarantees that all attacks in a certain class will be detected by the N-variant system as long as the normal equivalence property is satisfied. To establish the detection property, we need to know that any input that causes one variant to enter a compromised state must also cause some other variant to enter an alarm state. Because of the normal equivalence property, we can assume the variants all are in equivalent states before processing this input. Thus, we need to show:

$$\forall S \in Normal, 0 \leq v < N, S_v \text{ where } C_v (S_v) = S,$$
$$\forall p \in Inputs:$$
$$T_v (S_v, p) \in Compromised$$
$$\exists w \text{ such that } T_w (S_w, p) \in Alarm \text{ and } C_w (S_w) = S$$

If the detection property is established, we know that whenever one of the variants enters a compromised state, one of the variants must enter an alarm state. An ideal monitor would instantly detect the alarm state and prevent all the other variants from continuing. This would guarantee that the system never operates in a state in which any variant is compromised.

In practice, building such a monitor is impossible since we cannot keep the variants perfectly synchronized or detect alarm states instantly. However, we can approximate this behavior by delaying any external effects (including responses to the client) until all variants have passed a critical point. This keeps the variants loosely synchronized, and approximates the behavior of instantly terminating all other variants when one variant encounters an alarm state. It leaves open the possibility that a compromised variant could corrupt the state of other parts of the system (including the monitor and other variants) before the alarm state is detected. An implementation must use isolation mechanisms to limit this possibility.

4. Variations

Our framework works with any diversification technique that produces variants different enough to provide detection of a class of attack but similar enough to establish a normal equivalence property. The variation used to diversify the variants determines the attack class the N-variant system can detect. The detection property is defined by the class of attack we detect, so we will consider attack classes, such as attacks that involve executing injected instructions, rather than vulnerability classes such as buffer overflow vulnerabilities.

Next, we describe two variations we have implemented: address space partitioning and instruction set tagging. We argue (informally) that they satisfy both the normal equivalence property and the detection condition for important classes of attacks. The framework is general enough to support many other possible variations, which we plan to explore in future work. Other possible variations that could provide useful security properties include varying memory organization, file naming, scheduling, system calls, calling conventions, configuration properties, and the root user id.

4.1 Address Space Partitioning

The Introduction described an example variation where the address space is partitioned between two variants to disrupt attacks that rely on absolute addresses. This simple variation does not prevent all memory corruption attacks since some attacks depend only on relative addressing, but it does prevent all memory corruption attacks that involve direct references to absolute addresses. Several common vulnerabilities including format string [56, 54], integer overflow, and double-free [24] may allow an attacker to overwrite an absolute location in the target's address space. This opportunity can be exploited to give an attacker control of a process, for example, by modifying the Global Offset Table [24] or the .dtors segment of an ELF executable [48]. Regardless of the vulnerability exploited and the targeted data structure, if the attack depends on loading or storing to an absolute address it will be detected by our partitioning variants. Since the variation alters absolute addresses, it is necessary that the original program does not depend on actual memory addresses (for example, using the value of a pointer directly in a decision). Although it is easy to construct programs that do not satisfy this property, most sensible programs should not depend on actual memory addresses.

Detection. Suppose P_0 only uses addresses whose high bit is 0 and P_1 only uses addresses whose high bit is 1. We can map the normal state of P_0 and P_1 to equivalent states using the identity function for C_0 and a function that flips the high bit of all memory addresses for C_1 (to map onto the actual addresses used by P, more complex mapping functions may be needed). The transition functions, T_0 and T_1 are identical; the generated code is what makes things different since a different address will be referenced in the generated code for any absolute address reference. If an attack involves referencing an absolute address, the attacker must choose an address whose high bit is either a 0 or 1. If it is a 0, then P_0 may transition to a compromised state, but P_1 will transition to an alarm state when it attempts to access a memory address outside P_1's address space. In Unix systems, this alarm state is detected by the operating system as a segmentation fault. Conversely, if the attacker chooses an address whose high bit is 1, P_1 may be compromised but P_0 must enter an alarm state. In either case, the monitor detects the compromise and prevents any external state modifications including output transmission to the client.

Our detection argument relies on the assumption that the attacker must construct the entire address directly. For most scenarios, this assumption is likely to be valid. For certain vulnerabilities on platforms that are not byte-aligned, however, it may not be. If the attacker is able to overwrite an existing address in the program without overwriting the high bit, the attacker may be able to construct an address that is valid in both variants. Similarly, if an attacker can corrupt a value that is subsequently used with a transformed absolute address in an address calculation, the detection property is violated. As with relative attacks, this indirect memory attacks would not be detected by this variation.

Normal equivalence. We have two options for establishing the normal equivalence property: we can check it dynamically using the monitor, or we can prove it statically by analyzing the variants. A pure dynamic approach is attractive for security assurance because of its simplicity but impractical for performance-critical servers. The monitor would need to implement C_0 and C_1 and compute the canonical states of each variant at the end of each instruction execution. If the states match, normal equivalence is satisfied. In practice, however, this approach is likely to be prohibitively expensive. We can optimize the check by limiting the comparison to the subset of the execution state that may have changed and only checking the state after particular instructions, but the overhead of checking the states of the variants after every step will still be unacceptable for most services.

The static approach requires proving that for every possible normal state, all normal transitions result in equivalent states on the two variants. This property requires that no instruction in P can distinguish between the two variants. For example, if there were a conditional jump in P that depended on the high bit of the address of some variable, P_0 and P_1 would end up in different states after executing that instruction. An attacker could take advantage of such an opportunity to get the variants in different states such that an input that transitions P_0 to a compromised state does not cause P_1 to reach an alarm state. For example, if the divergence is used to put P_0 in a state where the next client input will be passed to a vulnerable string format call, but the next client input to P_1 is processed harmlessly by some other code, an attacker may be able to successfully compromise the N-variant system. A divergence could also occur if some part of the system is nondeterministic, and the operating environment does not eliminate this nondeterminism (see Section 5). Finally, if P is vulnerable to some other class of attack, such as code injection, an attacker may be able to alter the transition functions T_0 and T_1 in a way that allows the memory corruption attack to be exploited differently on the two variants to avoid detection (of course, an attacker who can inject code can already compromise the system in arbitrary ways).

In practice, it will not usually be possible to completely establish normal equivalence statically for real systems but rather we will use a combination of static and dynamic arguments, along with assumptions about the target service. A combination of static and dynamic techniques for checking equivalence may be able to provide higher assurance without the overhead necessary for full dynamic equivalence checking. Our prototype implementation checks equivalence dynamically at the level of system calls, but relies on informal static arguments to establish equivalence between them.

Implementation. To partition the address space, we vary the location of the application data and code segments. The memory addresses used by P_0 and P_1 are disjoint: any data address that is valid for P_0 is invalid for P_1, and vice versa. We use a linker script to create the two variants. Each variant loads both the code and data segments of the variants at different starting addresses from the other variant. To ensure that their sets of valid data memory addresses are disjoint, we use ulimit to limit the size of P_0's data segment so it cannot grow to overlap P_1's address space.

4.2 Instruction Set Tagging

Whereas partitioning the memory address space disrupts a class of memory corruption attacks, partitioning the instruction set disrupts code injection attacks. There are several possible ways to partition the instruction set.

One possibility would be to execute the variants on different processors, for example one variant could run on an x86 and the other on a PowerPC. Establishing the security of such an approach would be very difficult, however. To obtain the normal equivalence property we would need a way of mapping the concrete states of the different machines to a common state. Worse, to obtain the detection property, we would need to prove that no string of bits that corresponds to a successful malicious attack on one instruction set and a valid instruction sequence on the other instruction set. Although it is likely that most sequences of malicious x86 instructions contain an invalid PowerPC instruction, it is certainly possible for attackers to design instruction sequences that are valid on both platforms (although we are not aware of any programs that do this for the x86 and PowerPC, Sjoerd Mullender and Robbert van Renesse won the 1984 International Obfuscated C Code Contest with an entry that replaced main with an array of bytes that was valid machine code for both the Vax and PDP-11 but executed differently on each platform [35]).

Instead, we use a single instruction set but prepend a variant-specific tag to all instructions. The diversification transformation takes P and inserts the appropriate tag bit before each instruction to produce each variant.

Detection. The variation detects any attack that involves executing injected code, as long as the mechanism used to inject code involves injecting complete instructions. If memory is bit-addressable, an attacker could overwrite just the part of the instruction after the tag bit, thereby changing an existing instruction while preserving the original tag bit. If the attacker can inject the intended code in memory, and then have the program execute code already in the executable that transforms the injected memory (for example, by XORing each byte with a constant that is different in the two variants), then it is conceivable that an attacker could execute an indirect code injection attack where the code is transformed differently on the two variants before executing to evade the detection property. For all known realistic code injection attacks, neither of these is considered a serious risk.

Normal equivalence. The only difference between the two variants is the instruction tag, which has no effect on instruction execution. The variants could diverge, however, if the program examines its own instructions and makes decisions that depend on the tag. It is unlikely that a non-malicious program would do this. As with the memory partitioning, if the instruction tags are visible to the executing process an attacker might be able to make them execute code that depends on the instruction tags to cause the variants to diverge before launching the code injection attack on one of the variants. To prevent this, we need to store the tagged instructions in memory that is not readable to the executing process and remove the tags before those instructions reach the processor.

Implementation. To implement instruction set tagging, we use a combination of binary rewriting before execution and software dynamic translation during execution. We use Diablo [61, 22], a retargetable binary rewriting framework, to insert the tags. Diablo provides mechanisms for modifying an x86 binary in ELF format. We use these to insert the appropriate variant-specific tag before every instruction. For simplicity, we use a full byte tag even though a single bit would suffice for two variants. There is no need to keep the tags secret, just that they are different; we use 10101010 and 01010101 for the *A* and *B* variant tags.

At run-time, the tags are checked and removed before instructions reach the processor. This is done using Strata, a software dynamic translation tool [52, 53]. Strata and other software dynamic translators [4, 11] have demonstrated that it is possible to implement software dynamic translation without unreasonable performance penalty. In our experiments (Section 5), Strata's overhead is only a few percent. The Strata VM mediates application execution by examining and translating instructions before they execute on the host CPU. Translated instructions are placed in the fragment cache and then executed directly on the host CPU. Before switching to the application code, the Strata VM uses `mprotect` to protect critical data structures including the fragment cache from being overwritten by the application. At the end of a translated block, Strata appends trampoline code that will switch execution back to the Strata VM, passing in the next application PC so that the next fragment can be translated and execution will continue. We implement the instruction set tagging by extending Strata's instruction fetch module. The modified instruction fetch module checks that the fetched instruction has the correct tag for this variant; if it does not, a security violation is detected and execution terminates. Otherwise, it removes the instruction tag before placing the actual instruction in the fragment cache. The code executing on the host processor contains no tags and can execute normally.

5. Framework Implementation

Implementing an N-variant system involves generating variants such as those described in Section 4 as well as implementing the polygrapher and monitor. The trusted computing base comprises the polygrapher, monitor and mechanisms used to produce the variants, as well as any operating system functionality that is common across the variants. An overriding constraint on our design is that it be fully automated. Any technique that requires manual modification of the server to create variants or application-specific monitoring would impose too large a deployment burden to be used widely. To enable rapid development, our implementations are entirely in software. Hardware implementations would have security and performance advantages, especially in monitoring the instruction tags. Furthermore, placing monitoring as close as possible to the processor eliminates the risk that an attacker can exploit a vulnerability in the monitoring mechanism to inject instructions between the enforcement mechanism and the processor.

The design space for N-variant systems implementations presents a challenging trade-off between isolation of the variants, polygrapher, and monitor and the need to keep the variant processes synchronized enough to establish the normal equivalence property. The other main design decision is the granularity of the monitoring. Ideally, the complete state of each variant would be inspected after each instruction. For performance reasons, however, we can only observe aspects of the state at key execution points. Incomplete monitoring means that an attacker may be able to exploit a different vulnerability in the server to violate the normal equivalence property, thereby enabling an attack that would have otherwise been detected to be carried out without detection. For example, an attacker could exploit a race condition in the server to make the variants diverge in ways that are not detected by the monitor. Once the variants have diverged, the attacker can construct an input that exploits the vulnerability in one variant, but does not produce the detected alarm state on the other variants because they started from different states.

In our first proof-of-concept implementation, described in Section 5.1, we emphasized isolation and executed the variants on separate machines. This meant that any nondeterminism in the server program or aspects of the host state visible to the server program that differed between the machines could be exploited by an attacker to cause the processes to diverge and then allow a successful attack. It also meant the monitor only observed the outputs produced by the two variants that would be sent over the network. This enabled certain attacks to be detected, but meant a motivated attacker could cause the states to diverge in ways that were not visible from the output (such as corrupting server data) but still achieved the attacker's goals.

Our experience with this implementation led us to conclude that a general N-variant systems framework needed closer integration of the variant processes to prevent arbitrary divergences. We developed such a framework as a kernel modification that allows multiple variants to run on the same platform and normal equivalence to be established at system call granularity. This eliminates most causes of nondeterminism and improves the performance of the overall system. Section 5.2 describes our Linux kernel implementation, and Section 5.3 presents performance results running Apache variants on our system.

5.1 Proof-of-Concept Implementation

In our proof-of-concept implementation, the variants are isolated on separate machines and the polygrapher and monitor are both implemented by the nvd process running on its own machine. We used our implementation to protect both a toy server we constructed and Apache. In order for our approach to work in practice it is essential that no manual modification to the server source code is necessary. Hence, each server variant must execute in a context where it appears to be interacting normally with the client. We accomplish this by using divert sockets to give each variant the illusion that it is interacting directly with a normal client. To implement the polygrapher we use ipfw, a firewall implementation for FreeBSD [27] with a rule that redirects packets on port 80 (HTTP server) to our nvd process which adjusts the TCP sequence numbers to be consistent with the variant's numbering. Instead of sending responses directly to the client, the variant's responses are diverted back to nvd, which buffers the responses from all of the variants. The responses from P_0 are transmitted back to the client only if a comparably long response is also received from the other variants. Hence, if any variant crashes on a client input, the response is never sent to the client and nvd restarts the server in a known uncompromised state.

We tested our system by using it to protect a toy server we constructed with a simple vulnerability and Apache, and attempted to compromise those servers using previously known exploits as well as constructed exploits designed to attack a particular variant. Exploit testing does not provide any guarantees of the security of our system, of course, but it does demonstrate that the correct behavior happens under the tested conditions to increase our confidence in our approach and implementation. Our toy server contained a contrived format string vulnerability, and we developed an exploit that used that vulnerability to write to an arbitrary memory address. The exploit could be customized to work against either variation, but against the N-variant system both versions would lead to one of the variants crashing. The monitor detects the crash and prevents compromised outputs from reaching the client. We also tested an Apache server containing a vulnerable OpenSSL implementation (before 0.9.6e) that contained a buffer overflow vulnerability that a remote attacker could exploit to inject code [13]. When instruction set tagging is used, the exploit is disrupted since it does not contain the proper instruction tags in the injected code.

We also conducted some performance measurements on our 2-variant system with memory address partitioning. The average response latency for HTTP requests increased from 0.2ms for the unmodified server to 2.9ms for the 2-variant system.

The proof-of-concept implementation validated the N-variant systems framework concept, but did not provide a practical or secure implementation for realistic services. Due to isolation of the variants, various non-attack inputs could lead to divergences between the variants caused by differences between the hosts. For example, if the output web page includes a time stamp or host IP address, these would differ between the variants. This means false positives could occur when the monitor observes differences between the outputs for normal requests. Furthermore, a motivated attacker could take advantage of any of these differences to construct an attack that would compromise one of the variants without leading to a detected divergence.

5.2 Kernel Implementation

The difficulties in eliminating nondeterminism and providing finer grain monitoring with the isolated implementation, as well as its performance results, convinced us to develop a kernel implementation of the framework by modifying the Linux 2.6.11 kernel. In this implementation, all the variants run on the same platform, along with the polygrapher and monitor. We rely on existing operating system mechanisms to provide isolation between the variants, which execute as separate processes.

We modified the kernel data structures to keep track of variant processes and implemented wrappers around system calls. These wrappers implement the polygraphing functionality by wrapping input system calls so that when both variants make the same input system call, the actual input operation is performed once and the same data is sent to all variants. They provide the monitoring functionality by checking that all variants make the same call with equivalent arguments before making the actual system call.

This system call sharing approach removes nearly all of the causes of nondeterminism that were problematic in the proof-of-concept implementation. By wrapping the system calls, we ensure that variants receive identical results from all system calls. The remaining cause of nondeterminism is due to scheduling differences, in particular in handling signals. We discuss these limitations in Section 6.

In order to bring an N-variant system into execution we created two new system calls: n_variant_fork, and n_variant_execve. The program uses these system calls similarly to the way a shell uses fork/execve to bring processes into execution. The n_variant_fork system call forks off the variants, however instead of creating a single child process it creates one process per variant. The variants then proceed to call n_variant_execve, which will cause each of the variants to execute their own diversified binary of the server. Note that our approach requires no modification of an existing binary to execute it within an N-variant system; we simply invoke a shell command that takes the pathnames of variant binaries as parameters and executes n_variant_execve.

Next, we provide details on the system call wrappers that implement the polygraphing and monitoring. The Linux 2.6.11 kernel provides 267 system calls. We generalize them into three categories based on the type of wrapper they need: shared system calls, reflective system calls, and dangerous system calls.

Shared System Calls. For system calls that interact with external state, including I/O system calls, the wrapper checks that all variants make equivalent calls, makes the actual call once, and sends the output to all variants, copying data into each of the variants address space if necessary. Figure 2 shows pseudocode for a shared call, in this case the read system call. The actual wrappers are generated using a set of preprocessor macros we developed to avoid duplicating code. The first if statement checks whether this process is part of an N-variant system. If not, the system call proceeds normally. Hence, a single platform can run both normal and

```
ssize_t sys_read(int fd, const void *buf, size_t count) {
  if (!hasSibling (current)) { make system call normally } // not a variant process
  else {
    record that this variant process entered call
    if (!inSystemCall (current->sibling)) { // this variant is first
      save parameters
      sleep // sibling will wake us up
      get result and copy *buf data back into address space
      return result;
    } else if (currentSystemCall (current->sibling) == SYS_READ) { // this variant is second, sibling waiting
      if (parameters match) { // what it means to "match" depends on variation and system call
        perform system call
        save result and data in kernel buffer
        wake up sibling
        return result;
      } else { DIVERGENCE ERROR! } // sibling used different parameters
    } else { DIVERGENCE ERROR! } } } // sibling is in a different system call
```

Figure 2. Typical shared system call wrapper.

N-variant processes. If the process is a variant process, it records that it has entered this system call and checks if its sibling variant has already entered a system call. If it has not, it saves the parameters and sleeps until the other variant wakes it up. Otherwise, it checks that the system call and its parameters match those used by the first variant to make the system call. If they match, the actual system call is made. The result is copied into a kernel buffer, and the sibling variant process (which reached this system call first and went to sleep) is awoken. The sibling process copies the result from the kernel buffer back into its address space and continues execution.

Reflective System Calls. We consider any system call that observes or modifies properties of the process itself a *reflective* system call. For these calls, we need to ensure that all observations always return the same value regardless of which variant reaches the call first, and that all modifications to process properties are done equivalently on all variants. For observation-only reflective calls, such as getpid, we check that all variants make the same call, and then just make the call once for variant 0 and send the same result to all variants. This is done using wrappers similar to those for shared system calls, except instead of just allowing the last variant that reaches the call to make the actual system call we need to make sure that each time a reflective call is reached, it is executed for the same process.

Another issue is raised by the system calls that create child processes (sys_fork, sys_vfork, and sys_clone). The wrappers for these calls must coordinate each variant's fork and set up all the child processes as a child N-variant system before any of the children are placed on the run queue. These system calls return the child process' PID. We ensure that all the parents in the N-variant system get the same PID (the PID of variant 0's child), as with the process observation system calls.

The other type of reflective system call acts on the process itself. These system calls often take parameters given by the reflective observation system calls. In this case, we make sure they make the same call with the same parameters, but alter the parameters accordingly for each variant. For example, sys_wait4 takes a PID as an input. Each of the variants will call sys_wait4 with the same PID because they were all given the same child PID when they called sys_fork (as was required to maintain normal equivalence). However, each variant needs to clean up its corresponding child process within the child system. The wrapper for sys_wait4 modifies the PID value passed in and makes the appropriate call for each variant with its corresponding child PID. Similar issues arise with sys_kill, sys_tkill, and sys_waitpid.

Finally, we have to deal with two system calls that terminate a process: sys_exit and sys_exit_group. A terminating process does not necessarily go through these system calls, since it may terminate by crashing. To ensure that we capture all process termination events in an N-variant system we added a monitor inside the do_exit function within the kernel which is the last function all terminating processes execute. This way, if a process receives a signal and exits without going through a system call, we will still observe this and can terminate the other variants.

Dangerous System Calls. Certain calls would allow processes to break assumptions on which we rely. For example, if the process uses the execve system to run a new executable, this will escape the N-variant protections unless we can ensure that each variant executes a different executable that is diversified appropriately. Since it is unlikely we can establish this property, the execve wrapper just disables the system call and returns an error code. This did not pose problems for Apache, but might for other applications.

Other examples of dangerous system calls are those for memory mapping (old_mmap, sys_mmap2) which map a portion of a file into a process' address space. After a file is mapped into an address space, memory reads and writes are analogous to reads and writes from the file. This would allow an attacker to compromise one variant, and then use the compromised variant to alter the state of the uncompromised variants through the shared memory without detection, since no system call is necessary. Since many server applications (including Apache) use memory mapping, simply blocking these system calls is not an option. Instead, we place restrictions on them to allow only the MAP_ANONYMOUS and MAP_PRIVATE options with all permissions and to permit MAP_SHARED mappings as long as write permissions are not requested. This eliminates the communication channel between the variants, allowing memory mapping to be used safely by the variants. Apache runs even with these restrictions since it does not use other forms of memory mapping, but other solutions would be needed to support all services.

5.3 Performance

Table 1 summarizes our performance results. We measured the throughput and latency of our system using WebBench 5.0 [65], a web server benchmark using a variety of static web page requests. We ran two sets of

Configuration		1	2	3	4	5	6
Description		Unmodified Apache, unmodified kernel	Unmodified Apache, N-variant kernel	2-variant system, address partitioning	Apache running under Strata	Apache with instruction tags	2-variant system, instruction tags
Unsaturated	Throughput (MB/s)	2.36	2.32	2.04	2.27	2.25	1.80
	Latency (ms)	2.35	2.40	2.77	2.42	2.46	3.02
Saturated	Throughput (MB/s)	9.70	9.59	5.06	8.54	8.30	3.55
	Latency (ms)	17.65	17.80	34.20	20.30	20.58	48.30

Table 1. Performance Results.

experiments measuring the performance of our Apache server under unsaturated and saturated load conditions. In both sets, there was a single 2.2GHz Pentium 4 server machine with 1GB RAM running Fedora Core 3 (2.6.11 kernel) in the six different configurations shown in Table 1. For the first set of experiences, we used a single client machine running one WebBench client engine. For the load experiments, we saturated our server using six clients each running five WebBench client engines connected to the same networks switch as the server.

Configuration 1 is the baseline configuration: regular apache running on an unmodified kernel. Configuration 2 shows the overhead of the N-variant kernel on a normal process. In our experiments, it was negligible; this is unsurprising since the overhead is only a simple comparison at the beginning of each wrapped system call. Configuration 3 is a 2-variant system running in our N-variant framework where the two variants differ in the address spaces according to the partitioning scheme described in Section 4.1. For the unloaded server, the latency observed by the client increases by 17.6%. For the loaded server, the throughput decreases by 48% and the latency nearly doubles compared to the baseline configuration. Since the N-variant system executes all computation twice, but all I/O system calls only once, the overhead incurred reflects the cost of duplicating the computation, as well as the checking done by the wrappers. The overhead measured for the unloaded server is fairly low, since the process is primarily I/O bound; for the loaded server, the process becomes more compute-bound, and the approximately halving of throughput reflects the redundant computation required to run two variants.

The instruction tagging variation is more expensive because of the added cost of removing and checking the instruction tags. Configuration 4 shows the performance of Apache running on the normal kernel under Strata with no transformation. The overhead imposed by Strata reduces throughput by about 10%. The Strata overhead is relatively low because once a code fragment is in the fragment cache it does not need to be translated again the next time it executes. Adding the instruction tagging (Configuration 5) has minimal impact on throughput and latency. Configuration 6 shows the performance of a 2-variant system where the variants are running under Strata with instruction tag variation. The performance impact is more than it was in Configuration 3 because of the additional CPU workload imposed by the instruction tags. For the unloaded server, the latency increases 28% over the baseline configuration; for the saturated server, the throughput is 37% of the unmodified server's throughput.

Our results indicate that for I/O bound services, N-variant systems where the variation can be achieved with reasonable performance overhead, especially for variations such as the address space partitioning where little additional work is needed at run-time. We anticipate there being many other interesting variations of this type, such as file renaming, local memory rearrangement, system call number diversity, and user id diversity. For CPU-bound services, the overhead of our approach will remain relatively high since all computation needs to be performed twice. Multiprocessors may alleviate some of the problem (in cases where there is not enough load to keep the other processors busy normally). Fortunately, many important services are largely I/O-bound today and trends in processor and disk performance make this increasingly likely in the future.

6. Discussion

Our prototype implementation illustrates the potential for N-variant systems to protect vulnerable servers from important classes of attacks. Many other issues remain to be explored, including how our approach can be applied to other services, what variations can be created to detect other classes of attacks, how an N-variant system can recover from a detected attack, and how compositions of design and artificially diversified variants can provide additional security properties.

Applicability. Our prototype kernel implementation demonstrated the effectiveness of our approach using Apache as a target application. Although Apache is a representative server, there are a number of things other servers might do that would cause problems for our implementation. The version of Apache used in our experiments on uses the fork system call to create separate processes to handle requests. Each child process is run as an independent N-variant system. Some servers use user-level threading libraries where there are multiple threads within a single process invisible to our kernel monitor. This causes problems in an N-variant system, since the threads in the variants may interleave differently to produce different sequences of system calls (resulting in a false detection), or worse, interleave in a way that allows an attacker to exploit a race condition to carry out a successful attack without detection. One possible solution to this problem is to modify the thread scheduler to ensure that threads in the variants are scheduled identically to preserve synchronization between the variants.

The asynchronous property of process signals makes it difficult to ensure that all variants receive a signal at the exact same point in each of their executions. Although we can ensure that a signal is sent to all the variants at the same time, we cannot ensure that all the variants are exactly at the same point within their program at that time. As a result, the timing of a particular signal could cause divergent behavior in the variants if the code behaves differently depending on the exact point when the signal is received. This might cause the variants to diverge even though they are not under attack, leading to a false positive detection. As with user-level threads, if we modify the kernel to provide more control of the scheduler we could ensure that variants receive signals at the same execution points.

Another issue that limits application of our approach is the use of system calls we classified as dangerous such as execve or unrestricted use of mmap. With our current wrappers, a process that uses these calls is terminated since we cannot handle them safely in the N-variant framework. In some cases, more precise wrappers may allow these dangerous calls to be used safely in an N-variant system. Some calls, however, are inherently dangerous since they either break isolation between the variants or allow them to escape the framework. In these situations, either some loss of security would need to be accepted, or the application would need to be modified to avoid the dangerous system calls before it could be run as an N-variant system.

Other variations. The variations we have implemented only thwart attacks that require accessing absolute memory addresses or injecting code. For example, our current instruction tagging variation does not disrupt a *return-to-libc* attack (since it does not involve injecting code), and our address space partitioning variation provides no protection against memory corruption attacks that only use relative addressing. One goal for our future work is to devise variations that enable detection of larger classes of attack within the framework we have developed. We believe there are rich opportunities for incorporating different kinds of variation in our framework, although the variants must be designed carefully to ensure the detection and normal equivalence properties are satisfied. Possibilities include variations involving memory layout to prevent classes of relative addressing attacks, file system paths to disrupt attacks that depend on file names, scheduling to thwart race condition attacks, and data structure parameters to disrupt algorithmic complexity attacks [21].

Composition. Because of the need to satisfy the normal equivalence property, we cannot simply combine multiple variations into two variants to detect the union of their attack classes. In fact, such a combination risks compromising the security properties each variation would provide by itself. By combining variations more carefully, however, we can compose variants in a way that maintains the properties of the independent variations. To do this securely, we must ensure that, for each attack class we wish to detect, there is a pair of variants in the system that differs only in the transformation used to detect that attack class. This is necessary to ensure that for each variation, there is a pair of variants that satisfy the normal equivalence property for that variation but differ in the varied property. This approach can generalize to compose n binary variations using $n + 1$ variants. More clever approaches may be able to establish the orthogonality of certain variations to allow fewer variants without sacrificing normal equivalence.

Another promising direction is to combine our approach with design diversity approaches [46, 28, 62]. We could create a 3-variant system where two variants are Apache processes running on Linux hosts with controlled address space partitioning variation, and the third variant is a Windows machine running IIS. This would provide guaranteed detection of a class of low-level memory attacks through the two controlled variants, as well as probabilistic detection of attacks that exploit high-level application semantics through the design variants.

Recovery. Our modified kernel detects an attack when the system calls made by the variants diverge. At this point, one variant is in an alarm state (e.g., crashed), and the other variant is in a possibly compromised state. After detecting the attack, the monitor needs to restart the service in an uncompromised state. Note that the attack is always detected before any system call is executed for a compromised process; this means no external state has been corrupted. For a stateless server, the monitor can just restart all of the variants. For a stateful server, recovery is more difficult. One interesting approach is to compare the states of the variants after the attack is detected to determine the valid state. Depending on the variation used, it may be possible to recover a known uncompromised state from the state of the alarm variant, as well as to deduce an attack signature from the differences between the two variants' states. Another approach involves adding an extra *recovery variant* that maintains a known uncompromised state and can be used to restart the other variants after an attack is detected. The recovery variant could be the original *P*, except it would be kept behind the normal variants. The polygrapher would delay sending input to the recovery variant until all of the regular variants process it successfully. This complicates the wrappers substantially, however, and raises difficult questions about how far behind the recovery variant should be.

7. Conclusion

Although the cryptography community has developed techniques for proving security properties of cryptographic protocols, similar levels of assurance for system security properties remains an elusive goal. System software is typically too complex to prove it has no vulnerabilities, even for small, well-defined classes of vulnerabilities such as buffer overflows. Previous techniques for thwarting exploits of vulnerabilities have used ad hoc arguments and tests to support claimed security properties. Motivated attackers, however, regularly find ways to successfully attack systems protected using these techniques [12, 55, 58, 64].

Although many defenses are available for the particular attacks we address in this paper, the N-variant systems approach offers the promise of a more formal security argument against large attack classes and correspondingly higher levels of assurance. If we can prove that the automated diversity produces variants that satisfy both the normal equivalence and detection properties against a particular attack class, we can have a high degree of confidence that attacks in that class will be detected. The soundness of the argument depends on correct behavior of the polygrapher, monitor, variant generator and any common resources.

Our framework opens up exciting new opportunities for diversification approaches, since it eliminates the need for high entropy variations. By removing the reliance on keeping secrets and providing an architectural and associated proof framework for establishing security properties, N-variant systems offer potentially substantial gains in security for high assurance services.

Availability

Our implementation is available as source code from http://www.nvariant.org. This website also provides details on the different system call wrappers.

Acknowledgments

We thank Anil Somayaji for extensive comments and suggestions; Lorenzo Cavallaro for help with the memory partitioning scripts; Jessica Greer for assistance setting up our experimental infrastructure; Caroline Cox, Karsten Nohl, Nate Paul, Jeff Shirley, Nora Sovarel, Sean Talts, and Jinlin Yang for comments on the work and writing. This work was supported in part by grants from the DARPA Self-Regenerative Systems Program (FA8750-04-2-0246) and the National Science Foundation through NSF Cybertrust (CNS-0524432).

References

[1] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. *CCS* 2005.

[2] Starr Andersen. Changes to Functionality in Microsoft Windows XP Service Pack 2: Part 3: Memory Protection Technologies. *Microsoft TechNet*. August 2004.

[3] Algirdas Avizienis and L. Chen. On the Implementation of N-version Programming for Software Fault-Tolerance During Program Execution. *International Computer Software and Applications Conference*. 1977.

[4] Vasanth Bala, E. Duesterwald, S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. *ACM Programming Language Design and Implementation* (PLDI). 2000.

[5] Arash Baratloo, N. Singh, T. Tsai. Transparent Run-Time Defense against Stack Smashing Attacks. *USENIX Technical Conference*. 2000.

[6] Elena Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefanovic, D. Zovi. Intrusion Detection: Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. *CCS* 2003.

[7] Emery Berger and Benjamin Zorn. DieHard: Probabilistic Memory Safety for Unsafe Lan-

guages. *ACM Programming Language Design and Implementation* (PLDI), June 2006.

[8] Sandeep Bhatkar, Daniel DuVarney, and R. Sekar. Address Ofuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. *USENIX Security* 2003.

[9] Kenneth Birman. Replication and Fault Tolerance in the ISIS System. *10th ACM Symposium on Operating Systems Principles,* 1985.

[10] K. Birman, *Building Secure and Reliable Network Applications*, Manning Publications, 1996.

[11] Derek Bruening, Timothy Garnett, Saman Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. *International Symposium on Code Generation and Optimization.* 2003.

[12] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack*. Vol 0xa Issue 0x38. May 2000. http://www.phrack.org/phrack/56/p56-0x05

[13] CERT. *OpenSSL Servers Contain a Buffer Overflow During the SSL2 Handshake Process*. CERT Advisory CA-2002-23. July 2002.

[14] L. Chen and Algirdas Avizienis. N-Version Programming: A Fault Tolerance Approach to Reliability of Software Operation. *8th International Symposium on Fault-Tolerant Computing*. 1978.

[15] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. *USENIX Security* 2005.

[16] Marc Chérèque, David Powell, Philippe Reynier, Jean-Luc Richier, and Jacques Voiron. Active Replication in Delta-4. *22nd International Symposium on Fault-Tolerant Computing*. July 1992.

[17] Monica Chew and Dawn Song. *Mitigating Buffer Overflows by Operating System Randomization*. Tech Report CMU-CS-02-197. December 2002.

[18] George Coulouris, Jean Dollimore and Tim Kindberg. *Distributed Systems: Concepts and Design* (Third Edition). Addison-Wesley. 2001.

[19] Crispin Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. *USENIX Security* 1998.

[20] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. *USENIX Security* 2001.

[21] Scott Crosby and Dan Wallach. Denial of Service via Algorithmic Complexity Attacks. *USENIX Security* 2003.

[22] Bruno De Bus, Bjorn De Sutter, Ludo Van Put, D. Chanet, K. De Bosschere. Link-time Optimization of ARM Binaries. Language. *Compiler and Tool Support for Embedded Systems*. 2004.

[23] Nurit Dor, M. Rodeh, M. Sagiv. CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. *ACM Programming Language Design and Implementation*. June 2003.

[24] Jon Erickson**.** *Hacking: The Art of Exploitation*. No Starch Press. November 2003,

[25] Hiroaki Etoh. *GCC extension for protecting applications from stack-smashing attacks*. IBM, 2004. http://www.trl.ibm.com/projects/security/ssp

[26] Stephanie Forrest, Anil Somayaji, David Ackley. Building diverse computer systems. *6th Workshop on Hot Topics in Operating Systems*. 1997.

[27] The FreeBSD Documentation Project. *FreeBSD Handbook*, Chapter 24. 2005.

[28] Debin Gao, Michael Reiter, Dawn Song**.** Behavioral Distance for Intrusion Detection. *8th International Symposium on Recent Advances in Intrusion Detection*. September 2005.

[29] Daniel Geer, C. Pfleeger, B. Schneier, J. Quarterman, P. Metzger, R. Bace, P. Gutmann. *Cyberinsecurity: The Cost of Monopoly*. CCIA Technical Report, 2003.

[30] Eric Haugh and Matt Bishop. Testing C programs for buffer overflow vulnerabilities. *NDSS* 2003.

[31] David Holland, Ada Lim, and Margo Seltzer. An Architecture A Day Keeps the Hacker Away. *Workshop on Architectural Support for Security and Anti-Virus*. April 2004.

[32] D. Jewett. Integrity S2: A Fault-Tolerant Unix Platform. *17th International Symposium on Fault-Tolerant Computing Systems*. June 1991.

[33] Mark K. Joseph. *Architectural Issues in Fault-Tolerant, Secure Computing Systems*. Ph.D. Dissertation. UCLA Department of Computer Science, 1988.

[34] James Just, J. Reynolds, L. Clough, M. Danforth, K. Levitt, R. Maglich, J. Rowe. Learning Unknown Attacks – A Start*.* *Recent Advances in Intrusion Detection*. Oct 2002.

[35] Gaurav Kc, A. Keromytis, V. Prevelakis. Countering Code-injection Attacks with Instruction Set Randomization. *CCS* 2003.

[36] John Knight and N. Leveson. An Experimental Evaluation of the Assumption of Independence in Multi-version Programming. *IEEE Transactions on Software Engineering*, Vol 12, No 1. Jan 1986.

[37] Ken Knowlton. A Combination Hardware-Software Debugging System. *IEEE Transactions on Computers*. Vol 17, No 1. January 1968.

[38] Benjamin Kuperman, C. Brodley, H. Ozdoganoglu, T. Vijaykumar, A. Jalote. Detection and Prevention of Stack Buffer Overflow Attacks. *Communications of the ACM*, Nov 2005.

[39] David Larochelle and David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. *USENIX Security* 2001.

[40] Ruby Lee, D. Karig, J. McGregor, and Z. Shi. Enlisting Hardware Architecture to Thwart Malicious Code Injection. *International Conference on Security in Pervasive Computing*. March 2003.

[41] John McGregor, David Karig, Zhijie Shi, and Ruby Lee. A Processor Architecture Defense against Buffer Overflow Attacks. *IEEE International Conference on Information Technology: Research and Education*. August 2003.

[42] Sjoerd Mullender and Robbert van Renesse. The International Obfuscated C Code Contest Entry. 1984. http://www1.us.ioccc.org/1984/mullender.c

[43] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. *NDSS* 2005.

[44] Adam J. O'Donnell and H. Sethu. On Achieving Software Diversity for Improved Network Security using Distributed Coloring Algorithms. *CCS* 2004.

[45] Manish Prasad and T. Chiueh. A Binary Rewriting Defense against Stack-Based Buffer Overflow Attacks. *USENIX Technical Conference*. June 2003.

[46] James Reynolds, J. Just, E. Lawson, L. Clough, R. Maglich, K. Levitt. The Design and Implementation of an Intrusion Tolerant System. *Foundations of Intrusion Tolerant Systems* (OASIS). 2003.

[47] Michael Ringenburg and Dan Grossman. Preventing Format-String Attacks via Automatic and Efficient Dynamic Checking. *CCS* 2005.

[48] Juan Rivas. *Overwriting the .dtors Section*. Dec 2000. http://synnergy.net/downloads/papers/dtors.txt

[49] Olatunji Ruwase and Monica S. Lam. A Practical Dynamic Buffer Overflow Detector. *NDSS* 2004.

[50] Fred Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*. Dec 1990.

[51] Fred Schneider and L. Zhou. *Distributed Trust: Supporting Fault-Tolerance and Attack-Tolerance*, Cornell TR 2004-1924, January 2004.

[52] Kevin Scott and Jack W. Davidson. Safe Virtual Execution Using Software Dynamic Translation. *ACSAC*. December 2002.

[53] Kevin Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, M. L. Soffa. Retargetable and Reconfigurable Software Dynamic Translation. *International Symposium on Code Generation and Optimization*. March 2003.

[54] Scut / team teso. *Exploiting Format String Vulnerabilities*. March 2001.

[55] Hovav Shacham, M. Page, B. Pfaff, Eu-Jin Goh, N. Modadugu, Dan Boneh. On the effectiveness of address-space randomization. *CCS* 2004.

[56] Umesh Shankar, K. Talwar, J. Foster, D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. *USENIX Security* 2001.

[57] Stelios Sidiroglou, G. Giovanidis, A. Keromytis. A Dynamic Mechanism for Recovering from Buffer Overflow Attacks. *8th Information Security Conference*. September 2005.

[58] Ana Nora Sovarel, David Evans, Nathanael Paul. Where's the FEEB?: The Effectiveness of Instruction Set Randomization. *USENIX Security* 2005.

[59] Mark Stamp. Risks of Monoculture. *Communications of the ACM*. Vol 47, Number 3. March 2004.

[60] Karthik Sundaramoorthy, Z. Purser, E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. *Architectural Support for Programming Languages and Operating Systems* (ASPLOS). Nov 2000.

[61] Bjorn De Sutter and Koen De Bosschere. Introduction: Software techniques for Program Compaction. *Communications of the ACM*. Vol 46, No 8. Aug 2003.

[62] Eric Totel, Frédéric Majorczyk, Ludovic Mé. COTS Diversity Intrusion Detection and Application to Web Servers. *Recent Advances in Intrusion Detection*. September 2005.

[63] Timothy Tsai and Navjot Singh. *Libsafe 2.0: Detection of Format String Vulnerability Exploits*. Avaya Labs White Paper. February 2001.

[64] Nathan Tuck, B. Calder, and G. Varghese. Hardware and Binary Modification Support for Code Pointer Protection from Buffer Overflow**.** *International Symposium on Microarchitecture*. Dec 2004.

[65] VeriTest Corporation. *WebBench 5.0*. http://www.veritest.com/benchmarks/webbench

[66] John Viega, J. Bloch, T. Kohno, Gary McGraw. ITS4 : A Static Vulnerability Scanner for C and C++ Code. *ACSAC*. Dec 2000.

[67] David Wagner, J. Foster, E. Brewer, A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. *NDSS* 2000.

[68] D. Wilson. The STRATUS Computer System. *Resilient Computer Systems: Volume 1*. John Wiley and Sons, 1986. p. 208-231.

[69] Jun Xu, Z. Kalbarczyk, R. Iyer. Transparent Runtime Randomization for Security. *Symposium on Reliable and Distributed Systems*. October 2003.

[70] Yongguang Zhang, H. Vin, L. Alvisi, W. Lee, S. Dao. Heterogeneous Networking: a New Survivability Paradigm. *New Security Paradigms Workshop* 2001.

Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks

Wei Xu Sandeep Bhatkar R. Sekar
Department of Computer Science
Stony Brook University, Stony Brook, NY 11794-4400
`{weixu,sbhatkar,sekar}@cs.sunysb.edu`

Abstract

Policy-based confinement, employed in SELinux and specification-based intrusion detection systems, is a popular approach for defending against exploitation of vulnerabilities in benign software. Conventional access control policies employed in these approaches are effective in detecting privilege escalation attacks. However, they are unable to detect attacks that "hijack" legitimate access privileges granted to a program, e.g., an attack that subverts an FTP server to download the password file. (Note that an FTP server would normally need to access the password file for performing user authentication.) Some of the common attack types reported today, such as SQL injection and cross-site scripting, involve such subversion of legitimate access privileges. In this paper, we present a new approach to strengthen policy enforcement by augmenting security policies with information about the trustworthiness of data used in security-sensitive operations. We evaluated this technique using 9 available exploits involving several popular software packages containing the above types of vulnerabilities. Our technique sucessfully defeated these exploits.

1 Introduction

Information flow analysis (a.k.a. *taint analysis*) has played a central role in computer security for over three decades [1, 10, 8, 30, 25]. The recent works of [22, 28, 5] demonstrated a new application of this technique, namely, detection of exploits on contemporary software. Specifically, these techniques track the source of each byte of data that is manipulated by a program during its execution, and detect attacks that overwrite pointers with untrusted (i.e., attacker-provided) data. Since this is an essential step in most buffer overflow and related attacks, and since benign uses of programs should never involve outsiders supplying pointer values, such attacks can be detected accurately by these new techniques.

In this paper, we build on the basic idea of using fine-grained taint analysis for attack detection, but expand its scope by showing that the technique can be applied to detect a much wider range of attacks prevalent today. Specifically, we first develop a source-to-source transformation of C programs that can efficiently track information flows at runtime. We combine this information with security policies that can reason about the source of data used in security-critical operations. This combination turns out to be powerful for attack detection, and offers the following advantages over previous techniques:

- *Practicality.* The techniques of [28] and [5] rely on hardware-level support for taint-tracking, and hence cannot be applied to today's systems. TaintCheck [22] addresses this drawback, and is applicable to arbitrary COTS binaries. However, due to difficulties associated with static analysis (or transformation) of binaries, their implementation uses techniques based on a form of runtime instruction emulation [21], which causes a significant slowdown, e.g., Apache server response time increases by a factor of 10 while fetching 10KB pages. In contrast, our technique is much faster, increasing the response time by a factor of 1.1.
- *Broad applicability.* Our technique is directly applicable to programs written in C, and several other scripting languages (e.g., PHP, Bash) whose interpreters are implemented in C. Security-critical servers are most frequently implemented in C. In addition, PHP and similar scripting languages are common choices for implementing web applications, and more generally, server-side scripts.
- *Ability to detect a wide range of common attacks.* By combining expressive security policies with fine-grained taint information, our technique can address a broader range of attacks than previous techniques. Figure 1 shows the distribution of the 139 COTS software vulnerabilities reported in 2003 and 2004 in the most recent official CVE dataset (Ver. 20040901). Our technique is applicable for detecting exploitations of about 2/3rds of these vulnerabilities, including *buffer overflows, format-string attacks, SQL injection, cross-site scripting, command and shell-code injection,* and *directory traversal.* In contrast, previous approaches typically handled smaller attack classes,

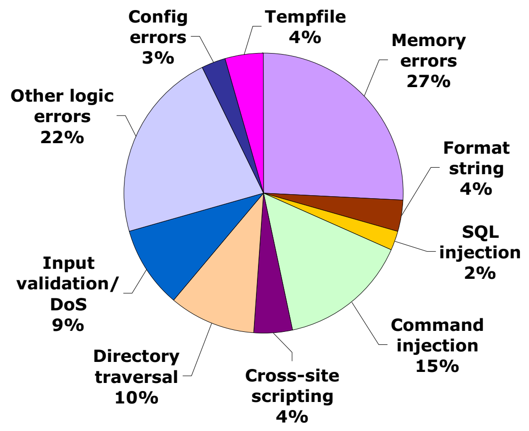


Figure 1: Breakdown of CVE software security vulnerabilities (2003 and 2004)

e.g., [7, 9, 2, 22, 28, 5] handle buffer overflows, [6] handles format string attacks, and [24, 23] handle injection attacks involving strings.

The focus of this paper is on the development of practical fine-grained dynamic taint-tracking techniques, and on illustrating how this information can be used to significantly strengthen conventional access control policies. For this purpose, we use simple taint-enhanced security policies. Our experimental evaluation, involving readily available exploits that target vulnerabilities in several popular applications, shows that the technique is effective against these exploits. Nevertheless, many of these policies need further refinement before they can be expected to stand up to skilled attackers. Section 7.2 discusses some of the issues in policy refinement, but the actual development of such refined policies is not a focus area of this paper.

We have successfully applied our technique to several medium to large programs, such as the PHP interpreter (300KLOC+) and `glibc`, the GNU standard C library (about 1MLOC). By leveraging the low-level nature of the C language, our implementation works correctly even in the face of memory errors, type casts, aliasing, and so on. At the same time, by exploiting the high-level nature of C (as compared to binary code), we have developed optimizations that significantly reduce the runtime overheads of fine-grained dynamic taint-tracking.

Approach Overview. Our approach consists of the following steps:

- *Fine-grained taint analysis:* The first step in our approach is a source-to-source transformation of C programs to perform runtime taint-tracking. Taint originates at input functions, e.g., a `read` or `recv` function used by a server to read network inputs. Input operations that return untrusted inputs are specified using *marking specifications* described in Section 4. In the transformed program, each byte of memory is associated with one bit (or more) of taint information. Logically, we can view the taint information as a bit-array `tagmap`, with `tagmap[a]` representing the taint information associated with the data at memory location a. As data propagates through memory, the associated taint information is propagated as well. Since taint information is associated with memory locations (rather than variables), our technique can ensure correct propagation of taint in the presence of memory errors, aliasing, type casts, and so on.
- *Policy enforcement:* This step is driven by *security policies* that are associated with security-critical functions. There are typically a small number of such functions, e.g., system calls such as `open` and `execve`, library functions such as `vfprintf`, functions to access external modules such as an SQL database, and so on. The security policy associated with each such function checks its arguments for "unsafe" content.

Organization of the Paper. We begin with motivating attack examples in Section 2. Section 3 describes our source-code transformation for fine-grained taint-tracking. Our policy language and sample policies are described in Section 4. The implementation of our approach is described in Section 5, followed by the experimental evaluation in Section 6. Section 7 discusses implicit information flows and security policy refinement. Section 8 presents related work. Finally, concluding remarks appear in Section 9.

2 Motivation for Taint-Enhanced Policies

In this section, we present several motivating attack examples. We conclude by pointing out the integral role played by taint analysis as well as security policies in detecting these attacks.

2.1 SQL and Command Injection. SQL injection is a common vulnerability in web applications. These server-side applications communicate with a web browser client to collect data, which is subsequently used to construct an SQL query that is sent to a back-end database. Consider the statement (written in PHP) for constructing an SQL query used to look up the price of an item specified by the variable `name`.

```
$cmd = "SELECT price FROM products WHERE
       name = '" . $name . "'"
```

If the value of `name` is assigned from an HTML form field that is provided by an untrusted user, then an SQL injection is possible. In particular, an attacker can provide the following value for `name`:

```
xyz'; UPDATE products SET price = 0 WHERE
name = 'OneCaratDiamondRing
```

With this value for `name`, `cmd` will take the value:

```
SELECT ...  WHERE name =
'$\boxed{\texttt{xyz'; UPDATE products SET price = 0 WHERE}}$
```

```
name = 'OneCaratDiamondRing'
```

Note that semicolons are used to separate SQL commands. Thus, the query constructed by the program will first retrieve the price of some item called `xyz`, and then set the price of another item called `OneCaratDiamondRing` to zero. This attack enables the attacker to purchase this item later for no cost.

Fine-grained taint analysis will mark every character in the query that is within the box as tainted. Now, a policy that precludes tainted control-characters (such as semicolons and quotes) or commands (such as `UPDATE`) in the SQL query will defeat the above attack. A more refined policy is described in Section 7.2.

Command injection attacks are similar to SQL injection: they involve untrusted inputs being used as to construct commands executed by command interpreters (e.g., `bash`) or the argument to `execve` system call.

2.2 Cross-Site Scripting (XSS). Consider an example of a bank that provides a "ATM locator" web page that customers can use to find the nearest ATM machine, based on their ZIP code. Typically, the web page contains a form that submits a query to the web site, which looks as follows:

```
http://www.xyzbank.com/findATM?zip=90100
```

If the ZIP code is invalid, the web site typically returns an error message such as:

```
<HTML> ZIP code not found: 90100 </HTML>
```

Note in the above output from the web server, the user-supplied string `90100` is reproduced. This can be used by an attacker to construct an XSS attack as follows. To do this, the attacker may send an HTML email to an unsuspecting user, which contains text such as:

```
To claim your reward, please click <a href="
  http://www.xyzbank.com/findATM?zip=
  <script%20src='http://www.attacker.com/
   malicious_script.js'></script>">here</a>
```

When the user clicks on this link, the request goes to the bank, which returns the following page:

```
<HTML> ZIP code not found:
  <script src='http://www.attacker.com/
   malicious_script.js'></script> </HTML>
```

The victim's browser, on receiving this page, will download and run Javascript code from the attacker's web site. Since the above page was sent from `http://www.xyzbank.com`, this script will have access to sensitive information stored on the victim computer that pertains to the bank, such as cookies. Thus, the above attack will allow cookie information to be stolen. Since cookies are often used to store authentication data, stealing them can allow attackers to perform financial transactions using victim's identity.

Fine-grained taint analysis will mark every character in the zip code value as tainted. Now the above cross-site scripting attack can be prevented by disallowing tainted `script` tags in the web application output.

2.3 Memory Error Exploits. There are many different types of memory error exploits, such as *stack-smashing, heap-overflows* and *integer overflows*. All of them share the same basic characteristics: they exploit bounds-checking errors to overwrite security-critical data, almost always a code pointer or a data pointer, with attacker-provided data. When fine-grained taint analysis is used, it will mark the overwritten pointer as tainted. Now, this attack can be stopped by a policy that prohibits dereferencing of tainted pointers.

2.4 Format String Vulnerabilities. The `printf` family of functions (which provide formatted printing in C) take a format string as a parameter, followed by zero or more parameters. A common misuse of these functions occurs when untrusted data is provided as the format string, as in the statement "`printf(s)`." If `s` contains an alphanumeric string, then this will not cause a problem, but if an attacker inserts format directives in `s`, then she can control the behavior of `printf`. In the worst case, an attacker can use the "`%n`" format directive, which can be used to overwrite a return address with attacker-provided data, and execute injected binary code.

When fine-grained taint analysis is used, the format directives (such as "`%n`") will be marked as tainted. The above attack can be then prevented by a taint-enhanced policy that disallows tainted format directives in the format string argument to the `printf` family of functions.

2.5 Attacks that "Hijack" Access Privileges. In this section, we consider attacks that attempt to evade detection by staying within the bounds of normal accesses made by an application. These attacks are also referred to as the confused deputy attacks [13].

Consider a web browser vulnerability that allows an attack (embedded within a web page) to upload an arbitrary file f owned by the browser user without the user's consent. Since the browser itself needs to access many of the user's files (e.g., cookies), a policy that prohibits access to f may prevent normal browser operations. Instead, we need a policy that can infer whether the access is being made during the normal course of an operation of the browser, or due to an attack. One way to do this is to take the taint information associated with the file name. If this file is accessed during normal browser operation, the file name would have originated within its program text or from the user. However, if the file name originated from a remote web site (i.e., an untrusted source), then it is likely to be an attack. Similar examples include attacks on (a) P2P applications to upload (i.e., steal) user files, and (b) FTP servers to download sensitive files such as the password file that are normally accessed by the server.

A variant of the above scenario occurs in the context of *directory traversal* attacks, where an attacker attempts to access files outside of an authorized directory, e.g., the document root in the case of a web server. Typically, this is done by including "`..`" characters in file names to ascend above the document root. In case the victim application already incorporates checks for "`..`" characters, attacker may attempt to evade this check by replacing "`.`" with its hexadecimal or Unicode representation, or by using various escape sequences. A taint-enhanced policy can be used to selectively enforce a more restrictive policy on file access when the file name is tainted, e.g., accesses outside of the document root directory may be disallowed. Such a policy would not interfere with the web server's ability to access other files, e.g., its access log or error log.

The key point about all attacks discussed in this section is that conventional access control policies cannot detect them. This is because the attacks do not stray beyond the set of resources that are normally accessed by a victim program. However, taint analysis provides a clue to *infer the intended use of an access.* By incorporating this inferred intent in granting access requests, taint-enhanced policies can provide better discrimination between attacks and legitimate uses of the privileges granted to a victim application.

2.6 Discussion. The examples discussed above bring out the following important points:

- *Importance of fine-grained taint information.* If we used coarser granularity for taint-tracking, e.g., by marking a program variable as tainted or untainted, we would not be able to detect most of the attacks described above. For instance, in the case of SQL injection example, the variable `cmd` containing the SQL query will always be marked as tainted, as it derives part of its value from an untrusted variable `name`. As a result, we cannot distinguish between legitimate uses of the web application, when `name` contains an alphanumeric string, from an attack, when `name` contains characters such as the semicolon and SQL commands. A similar analysis can be made in the case of stack-smashing and format-string attacks, cross-site scripting, directory traversal, and so on.
- *Need for taint-enhanced policies.* It is not possible to prevent these attacks by enforcing conventional access control policies. For instance, in the SQL injection example, one cannot use a policy that uniformly prevents the use of semicolons and SQL commands in `cmd`: such a policy would preclude any use of the database, and cause the web application to fail. Similarly, in the memory error example, one cannot have a working program if all control transfers through pointers are prevented. Finally, the examples in Section 2.5 were specifically chosen to illustrate the need for combining taint information into policies.

Another point to be made in this regard is that attacks are not characterized simply by the presence or absence of tainted information in arguments to security-critical operations. Instead, it is necessary to develop policies that govern the manner in which tainted data is used in these arguments.

3 Transformation for Taint Tracking

There are three main steps in taint-enhanced policy enforcement: (i) *marking*, i.e., identifying which external inputs to the program are untrusted and should be marked as tainted, (ii) *tracking* the flow of taint through the program, and (iii) *checking* inputs to security-sensitive operations using taint-enhanced policies. This section discusses tracking, which is implemented using a source-to-source transformation on C programs. The other two steps are described in Section 4.

3.1 Runtime Representation of Taint

Our technique tracks taint information at the level of bytes in memory. This is necessary to ensure accurate taint-tracking for type-unsafe languages such as C, since the approach can correctly deal with situations such as out-of-bounds array writes that overwrite adjacent data. A one-bit taint-tag is used for each byte of memory, with a '0' representing the absence of taint, and a '1' representing the presence of taint. A bit-array `tagmap` stores taint information. The taint bit associated with a byte at address a is given by `tagmap[a]`.

3.2 Basic Transformation

The source-code transformation described in this section is designed to track *explicit information flows* that take place through assignments and arithmetic and bit-operations. Flows that take place through conditionals are addressed in Section 7.1. It is unusual in C programs to have boolean-valued variables that are assigned the results of relational or logical operations. Hence we have not considered taint propagation through such operators in this paper. At a high-level, explicit flows are simple to understand:

- the result of an arithmetic/bit expression is tainted if any of the variables in the expression is tainted;
- a variable x is tainted by an assignment $x = e$ whenever e is tainted.

Specifically, Figure 2 shows how to compute the taint value $T(E)$ for an expression E. Figure 3 defines how a statement S is transformed, and uses the definition of $T(E)$. When describing the transformation rules, we

E	$T(E)$	Comment
c	0	Constants are untainted
v	$tag(\&v, sizeof(v))$	$tag(a, n)$ refers to n bits starting at $tagmap[a]$
$\&E$	0	An address is always untainted
$*E$	$tag(E, sizeof(*E))$	
$(cast)E$	$T(E)$	Type casts don't change taint.
$op(E)$	$T(E)$ 0	for arithmetic/bit op otherwise
$E_1\ op\ E_2$	$T(E_1) \mid\mid T(E_2)$ 0	for arithmetic/bit op otherwise

Figure 2: Definition of taint for expressions

S	$Trans(S)$
$v = E$	$v = E;$ $tag(\&v, sizeof(v)) = T(E);$
$S_1; S_2$	$Trans(S_1); Trans(S_2)$
$if\ (E)\ S_1$ $else\ S_2$	$if\ (E)\ Trans(S_1)$ $else\ Trans(S_2)$
$while\ (E)\ S$	$while\ (E)\ Trans(S)$
$return\ E$	$return\ (E, T(E))$
$f(a)\ \{\ S\ \}$	$f(a, ta)\ \{$ $tag(\&a, sizeof(a)) = ta; Trans(S)\}$
$v = f(E)$ $v = (*f)(E)$	$(v, tag(\&v, sizeof(v))) = f(E, T(E))$ $(v, tag(\&v, sizeof(v))) = (*f)(E, T(E))$

Figure 3: Transformation of statements for taint-tracking

use a simpler form of C (e.g. expressions have no side effects). In our implementation, we use the CIL [19] toolkit as the C front end to provide the simpler C form that we need.

The transformation rules are self-explanatory for most part, so we explain only the function-call related transformations. Consider a statement $v = f(E)$, where f takes a single argument. We introduce an additional argument ta in the definition of f so that the taint tag associated with its (single) parameter could be passed in. ta is explicitly assigned as the taint value of a at the beginning of f's body. (These two steps are necessary since the C language uses call-by-value semantics. If call-by-reference were to be used, then neither step would be needed.) In a similar way, the taint associated with the return value has to be explicitly passed back to the caller. We represent this in the transformation by returning a pair of values as the return value. (In our implementation, we do not actually introduce additional parameters or return values; instead, we use a second stack to communicate the taint values between the caller and the callee.) It is straight-forward to extend the transformation rules to handle multi-argument functions.

We conclude this section with a clarification on our notion of soundness of taint information. Consider any variable x at any point during any execution of a transformed program, and let a denote the location of this variable. If the value stored at a is obtained from any tainted input through assignments and arithmetic/bit operations, then `tagmap[a]` should be set. Note that by referring to the location of x rather than its name, we require that taint information be accurately tracked in the presence of memory errors. To support this notion of soundness, we needed to protect the `tagmap` from corruption, as described in Section 3.4.

3.3 Optimizations

The basic transformation described above is effective, but introduces high overheads, sometimes causing a slowdown by a factor of 5 or more. To improve performance, we have developed several interesting runtime and compile-time optimizations that have reduced overheads significantly. More details about the performance can be found in Section 6.4.

3.3.1 Runtime Optimizations In this section, we describe optimizations to the runtime data structures.

Use of 2-bit taint values. In the implementation, accessing of taint-bits requires several bit-masking, bit-shifting and unmasking operations, which degrade performance significantly. We observed that if 2-bit taint tags are used, the taint value for an integer will be contained within a single byte (assuming 32-bit architecture), thereby eliminating these bit-level operations. Since integer assignments occur very frequently, this optimization is quite effective.

This approach does increase the memory requirement for `tagmap` by a factor of two, but on the other hand, it opens up the possibility of tracking richer taint information. For instance, it becomes possible to associate different taint tags with different input sources and track them independently. Alternatively, it may be possible to use the two bits to capture "degree of taintedness."

Allocation of `tagmap`. Initially, we used a global variable to implement `tagmap`. But the initialization of this huge array (1GB) that took place at the program start incurred significant overheads. Note that tag initialization is warranted only for static data that is initialized at program start. Other data (e.g., stack and heap data) should be initialized (using assignments) before use in a correctly implemented program. When these assignments are transformed, the associated taint data will also be initialized, and hence there is no need to initialize such taint data in the first place. So, we allocated `tagmap` dynamically, and initialized only the locations corresponding to static data. By using `mmap` for this allocation, and by performing the allocation at a fixed address that is unused in Linux (our implementation platform), we ensured that runtime accesses to `tagmap` elements will be no more expensive than that of a statically allocated array (whose base address is also determined at compile-time).

The above approach reduced the startup overheads, but the mere use of address space seemed to tie up OS resources such as page table entries, and significantly increased time for `fork` operations. For programs such as shells that fork frequently, this overhead becomes unacceptable. So we devised an *incremental allocation* technique that can be likened to user-level page-fault handling. Initially, `tagmap` points to 1GB of address space that is unmapped. When any access to `tagmap[i]` is made, it results in a UNIX signal due to a memory fault. In the transformed program, we introduce code that intercepts this signal. This code queries the operating system to determine the faulting address. If it falls within the range of `tagmap`, a chunk of memory (say, 16KB) that spans the faulting address is allocated using `mmap`. If the faulting address is outside the range of `tagmap`, the signal is forwarded to the default signal handler.

3.3.2 Compile-time Optimizations

Use of local taint tag variables. In most C programs, operations on local variables occur much more frequently than global variables. Modern compilers are good at optimizing local variable operations, but due to possible aliasing, most such optimizations cannot be safely applied to global arrays. Unfortunately, the basic transformation introduces one operation on a global array for each operation on a local variable, and this has the effect of more than doubling the runtime of transformed programs. To address this problem we modified our transformation so that it uses local variables to hold taint information for local variables, so that the code added by the transformer can be optimized as easily as the original code.

Note, however, that the use of local tag variables would be unsound if aliasing of a local variable is possible. For example, consider the following code snippet:

```
int x; int *y = &x;
x = u; *y = v;
```

If `u` is untainted and `v` is tainted, then the value stored in `x` should be tainted at the end of the above code snippet. However, if we introduced a local variable, say, `tag_x`, to store the taint value of `x`, then we cannot make sure that it will get updated by the assignment to `*y`.

To ensure that taint information is tracked accurately, our transformation uses local taint tag variables only in those cases where no aliasing is possible, i.e., the optimization is limited to simple variables (not arrays) whose address is never taken. However, this alone is not enough, as aliasing may still be possible due to memory errors. For instance, a simple variable `x` may get updated due to an out-of-bounds access on an adjacent array, say, `z`. To eliminate this possibility, we split the runtime stack into two stacks. The main stack stores only simple variables whose addresses are never taken. This stack is also used for call-return. All other local variables are stored in the second stack, also called *shadow stack*.

The last possibility for aliasing arises due to pointer-forging. In programs with possible memory errors, a pointer to a local variable may be created. However, with the above transformation, any access to the main stack using a pointer indicates a memory error. We show how to implement an efficient mechanism to prevent access to some sections of memory in the transformed program. Using this technique, we prevent all accesses to the main stack except using local variable names, thus ensuring that taint information can be accurately tracked for the variables on the main stack using local taint tag variables.

Intra-procedural dependency analysis is performed to determine whether a local variable can ever become tainted, and to remove taint updates if it cannot. Note that a local variable can become tainted *only if* it is involved in an assignment with a global variable, a procedure parameter, or another local variable that can become tainted. Due to aliasing issues, this optimization is applied only to variables on the main stack.

3.4 Protecting Memory Regions

To ensure accurate taint-tracking, it is necessary to preclude access to certain regions of memory. Specifically, we need to ensure that the `tagmap` array itself cannot be written by the program. Otherwise, `tagmap` may be corrupted due to programming errors, or even worse, a carefully crafted attack may be able to evade detection by modifying the `tagmap` to hide the propagation of tainted data. A second region that needs to be protected is the main stack. Third, it would be desirable to protect memory that should not directly be accessed by a program, e.g., the GOT. (Global Offset Table is used for dynamic linking, but there should not be any reference to the GOT in the C code. If the GOT is protected in this manner, that would rule out attacks based on corrupting a function pointer in the GOT.)

The basic idea is as follows. Consider an assignment to a memory location a. Our transformation ensures that an access to `tagmap[a]` will be made before a is accessed. Thus, in order to protect a range of memory locations l—h, it is enough if we ensure that `tagmap[l]` through `tagmap[h]` will be unmapped. This is easy to do, given our incremental approach to allocation of `tagmap`. Now, any access to addresses l through h will result in a memory fault when the corresponding `tagmap` location is accessed.

Note that l and h cannot be arbitrary: they should fall on a 16K boundary, if the page size is 4KB and if 2 bit tainting is used. This is because `mmap` allocates memory blocks whose sizes are a multiple of a page size. This alignment requirement is not a problem for `tagmap`, since we can align it on a 16K boundary. For the main

Attack Type	Policy	Comment
Control-flow hijack	**jmp**(*addr*) | *addr* **matches** $(any+)^t \rightarrow term()$	Tainted values cannot be used as a target of control transfer
Format string	Format =`"%[^%]"` vfprintf(*fmt*) |*fmt* **matches** $any*$ (Format)^T $any* \rightarrow reject()$	Format directives (e.g.`%n`) should not be tainted
Directory traversal	DirTraversalModifier = `".."` file_function(*path*) = *open*(*path*,) || *unlink*(*path*) || ... file_function(*path*) | *path* **matches** $any*$ (DirTraversalModifier)^T $any*$ && *escapeRootDir*(*path*) $\rightarrow reject()$	If *path* contains tainted directory traversal strings (e.g. “..”), then the real path of *path* should not go outside the top level directories that are allowed to be accessed by the program, e.g. DocumentRoot and cgi-bin for httpd
Cross-site scripting	ScriptTag = `"<script"` | ... html_print_function(*str*) | *str* **matches** (StrIdNum|Delim) $*$ (ScriptTag)^T $any* \rightarrow reject()$	No tainted script tags (e.g. *script*) should be output to HTML.
SQL injection	SqlMetachar = `"'"` | `";"` | `"/*"` | ... sql_query_function(*query*) | *query* **matches** (StrIdNum|Delim) $*$ (SqlMetachar)^T $any*$ $\rightarrow reject()$	SQL query string should not contain tainted SQL meta-chars
Shell command injection	ShellMetachar = `";"` | `"&&"` | ... shell_command_function(*cmd*) | *cmd* **matches** (StrIdNum|Delim) $*$ (ShellMetachar)^T $any*$ $\rightarrow reject()$	*cmd* argument of *system* or *popen* should not contain tainted shell meta-chars

Figure 4: Illustrative security policies

stack, a potential issue arises because the bottom of the stack holds environment variables and command-line arguments that are arrays. To deal with this problem, we first introduce a gap in the stack in `main` so that its top is aligned on a 16K boundary. The region of main stack above this point is protected using the above mechanism. This means that it is safe to use local tag variables in any function except `main`.

4 Marking and Policy Specification

4.1 Marking Trusted and Untrusted Inputs

Marking involve associating taint information with all the data coming from external sources. If all code, including libraries, is transformed, then marking needs to be specified for system calls that return inputs, for environment variables and command-line arguments. (If some libraries are not transformed, then marking specifications may be needed for untransformed library functions that perform inputs.) Note that we can treat command-line arguments and environment variables as arguments to `main`. Thus, marking specifications can, in every case, be associated with a function call.

Marking actions are specified using BMSL (Behavior Monitoring Specification Language) [29, 3], an event-based language that is designed to support specification of security policies and behaviors. BMSL specifications consist of rules of the form $event\_pattern \longrightarrow action$. We use BMSL in a simplified way in this paper — in particular, $event\_pattern$ will be of the form $event \mid condition$, where $event$ identifies a function. When this function returns, and (the optional) *condition* holds, $action$ will be executed. The event corresponding to a function will take an additional argument that captures the return value from the function. Both the *condition* and the *action* can use external functions (written in C or C++). Moreover, the *action* can include arithmetic and logical operations, as well as if-then-else. Consider the following example:

```
read(fd, buf, size, rv)|(rv > 0) →
  if (isNetworkEndpoint(fd))
      taint_buffer(buf, rv);
  else untaint_buffer(buf, rv);
```

This rule states that when the `read` function returns, the `buf` argument will be tainted, based on whether the read was from a network or not, as determined by the external function `isNetworkEndpoint`. The actual tainting is done using two support functions `taint_buffer` and `untaint_buffer`.

Note that every input action needs to have an associated marking rule. To reduce the burden of writing many rules, we provide default rules for all system calls that untaint the data returned by each system call. Specific rules that override these default rules, such as the rule given above, can then be supplied by a user.

4.2 Specifying Policies

Security policies are also written using BMSL, but these rules are somewhat different from the marking rules. For a policy rule involving a function f, its *condition* component is examined immediately *before* any invocation of f. To simplify the policy specification, *abstract events* can be defined to represent a set of functions that share the same security policy. (Abstract events can be thought of as macros.)

The definition of *condition* is also extended to support regular-expression based pattern matching, using the keyword `matches`. We use *taint-annotated* regular expressions defined as follows. A tainted regular expression is obtained for a normal regular expression by attaching a superscript t, T or u. A string s will match a taint-annotated regular expression r^t provided that s matches r, and at least one of the characters in s is tainted. Similarly, s will match r^T provided *all* characters in s are tainted. Finally, s will match r^u provided none of the characters in s are tainted.

The predefined pattern any matches any single character. Parentheses and other standard regular expression operators are used in the usual way. Moreover, taint-annotated regular expressions can be named, and the name can be reused subsequently, e.g., `StrIdNum` used in many sample policy rules is defined as:

```
StrIdNum = String | Id | Num
```

where `String`, `Id` and `Num` denote named regular expressions that correspond respectively to strings, identifiers and numbers. Also, `Delim` denotes delimiters.

Figure 4 shows the examples of a few simple policies to detect various attacks. The *action* component of these policies make use of two support functions: $term()$ terminates the program execution, while $reject()$ denies the request and returns with an error.

For the control-flow hijack policy, we use a special keyword `jmp` as a function name, as we need some special way to capture low-level control-flow transfers that are not exposed as a function call in the C language. The policy states that if any of the bytes in the target address are tainted, then the program should be terminated.

For format string attacks, we only define a policy for `vfprintf`, because `vfprintf` is the common function used internally to implement all other `printf` family of functions. All format directives in a format string begin with a "`%`", and are followed by a character other than "`%`". (The sequence "`%%`" will simply print a "`%`", and hence can be permitted in the format string.)

Example policies to detect four other attacks, namely, directory traversal, cross-site scripting, SQL injection and shell command injection are also shown in Figure 4. The comments associated with the policies provide an intuitive description of the policy. These policies were able to detect all of the attacks considered in our evaluation, but we do not make any claim that the policies are good enough to detect all possible attacks in these categories. A discussion of how skilled attackers may evade some of these policies, and some directions for refining policies to stand up to such attacks, can be found in Section 7.2. The main strength of the approach presented in this paper is that the availability of fine-grained taint information makes it possible for a knowledgeable system administrator to develop such refined policies.

5 Implementation

We have implemented the program transformation technique described in Section 3. The transformer consists of about 3,600 lines of Objective Caml code and uses the CIL [19] toolkit as the front end to manipulate C constructs. Our implementation currently handles `glibc` (containing around 1 million LOC) and several other medium to large applications. The complexity and size of `glibc` demonstrated that our implementation can handle "real-world" code. We summarize some of the key issues involved in our implementation.

5.1 Coping with Untransformed Libraries

Ideally, all the libraries used by an application will be transformed using our technique so as to enable accurate taint tracking. In practice, however, source code may not be available for some libraries, or in rare cases, some functions in a library may be implemented in an assembly language. One option with such libraries is to do nothing at all. Our implementation is designed to work in these cases, but clearly, the ability to track information flow via untransformed functions is lost. To overcome this problem, our implementation offers two features. First, it produces warnings when a certain function could not be transformed. This ensures that inaccuracies will not be introduced into taint tracking without explicit knowledge of the user. When the user sees this warning, she may decide that the function in question performs largely "read" operations, or will never handle tainted data, and hence the warning can safely be ignored. If not, then our implementation supports *summarization functions* that specify how taint information is propagated by a function. For instance, we use the following summarization function for the `memcpy`. Summarization functions are also specified in BMSL, and use support functions to copy taint information. A summarization function for f would be invoked in the transformed code when f returns.

```
memcpy(dest, src, n) →
  copy_buffer_tagmap(dest, src, n);
```

So far, we had to write summarization functions for two `glibc` functions that are written in assembly and copy data, namely, `memcpy` and `memset`. In addition, `gcc` replaces calls to some functions such as `strcpy` and `strdup` with its own code, necessitating an additional 13 summarization functions.

5.2 Injecting Marking, Checking and Summarization Code into Transformed Programs

In our current implementation, the marking specifications, security policies, and summarization code associated with a function f are all injected into the transformed program by simply inlining (or explicitly call-

CVE#	Program	Language	Attack type	Attack description
CAN-2003-0201	samba	C	Stack smashing	Buffer overflow in call_trans2open function
CVE-2000-0573	wu-ftpd	C	Format string	via SITE EXEC command
CAN-2005-1365	pico server	C	Directory traversal	Command execution via URL with multiple leading "/" characters and ".."
CAN-2003-0486	phpBB 2.0.5	PHP	SQL injection	via topic_id parameter
CAN-2005-0258	phpBB 2.0.5	PHP	Directory traversal	Delete arbitrary file via ".." sequences in avatarselect parameter
CAN-2002-1341	SquirrelMail 1.2.10	PHP	Cross site scripting	Insert script via the mailbox parameter in read_body.php
CAN-2003-0990	SquirrelMail 1.4.0	PHP	Command injection	via meta-character in"To:" field
CAN-2005-1921	PHP XML-RPC	PHP	Command injection	Eval injection
CVE-1999-0045	nph-test-cgi	BASH	Shell meta-character expansion	using '*' in `$QUERY_STRING`

Figure 5: Attacks used in effectiveness evaluation

ing) the relevant code before or after the call to f. In the future, we anticipate these code to be decoupled from the transformation, and be able to operate on binaries using techniques such as library interposition. This would enable a site administrator to alter, refine or customize her notions of "trustworthy input" and "dangerous arguments" without having access to the source code.

6 Experimental Evaluation

The main goal of our experiments was to evaluate attack detection (Section 6.1), and runtime performance (Section 6.4). False positives and false negatives are discussed in Sections 6.2 and 6.3.

6.1 Attack Detection

Table 5 shows the attacks used in our experiments. These attacks were chosen to cover the range of attack categories we have discussed, and to span multiple programming languages. Wherever possible, we selected attacks on widely-used applications, since it is likely that obvious security vulnerabilities in such applications would have been fixed, and hence we are more likely to detect more complex attacks.

In terms of marking, all inputs read from network (using `read`, `recv` and `recvfrom`) were marked as tainted. Since the PHP interpreter is configured as a module for Apache, the same technique works for PHP applications as well. Network data is tainted when it is read by Apache, and this information propagates through the PHP interpreter, and in effect, through the PHP application as well. The policies used in our attack examples were already discussed in Section 4.

To test our technique, we first downloaded the software packages shown in Figure 5. We downloaded the exploit code for the attacks, and verified that they worked as expected. Then we used transformed C programs and interpreters with policy checking enabled, and verified that each one of the attacks were prevented by these policies without raising false alarms.

Network Servers in C.

- `wu-ftpd` versions 2.6.0 and lower have a format string vulnerability in `SITE EXEC` command that allows arbitrary code execution. The attack is stopped by the policy that the format directive `%n` in a format string should not be tainted.
- `samba` versions 2.2.8 and lower have a stack-smashing vulnerability in processing a type of request called "transaction 2 open." No policy is required to stop this attack — the stack-smashing step ends up corrupting some data on the shadow stack rather than the main stack, so the attack fails.

  If we had used an attack that uses a heap overflow to overwrite a GOT entry (which is common with heap overflows), this too would be detected without the need for any policies due to the technique described in Section 3.4 for preventing the GOT from being directly accessed by the C code. The reasoning is that before the injected code gets control, the GOT entry has to be clobbered by the existing code in the program. The instrumentation in the clobbering code will cause a segmentation fault because of the protection of the GOT, and hence the attack will be prevented. Note that the GOT is normally used by the PLT (Procedure Linkage Table) code that is in the assembly code automatically added by the compiler, and is not in the C source code, so a normal GOT access will not be instrumented with checks on taint tags, and hence will not lead to a memory fault.

  If the attack corrupted some other function pointer, then the "jmp" policy would detect the use of tainted data in jump target and stop the attack.
- `Pico HTTP Server (pServ)` versions 3.2 and

lower have a directory traversal vulnerability. The web server does include checks for the presence of "`..`" in the file name, but allows them as long as their use does not go outside the `cgi-bin` directory. To determine this, `pServ` scans the file name left-to-right, decrementing the count for each occurrence of "`..`", and incrementing it for each occurrence of "`/`" character. If the counter goes to zero, then access is disallowed. Unfortunately, a file name such as `/cgi-bin////../../bin/sh` satisfies this check, but has the effect of going outside the `/cgi-bin` directory. This attack is stopped by the directory traversal policy shown in Section 4.

Web Applications in PHP.

- `phpBB2` *SQL injection* vulnerability in (version 2.0.5 of) `phpBB`, a popular electronic bulletin board application, allows an attacker to steal the MD5 password hash of another user. The vulnerable code is:

```
$sql="SELECT p.post_id FROM ... WHERE ...
      AND p.topic_id = $topic_id AND ..."
```

  Normally, the user-supplied value for the variable `topic_id` should be a number, and in that case, the above query works as expected. Suppose that the attacker provides the following value:

```
-1 UNION SELECT ord(substring(user_password,
5,1)) FROM phpbb_users WHERE userid=3/*
```

  This converts the SQL query into a union of two `SELECT` statements, and comments out (using "`/*`") the remaining part of the original query. The first `SELECT` returns an empty set since `topic_id` is set to `-1`. As a result, the query result equals the value of the `SELECT` statement injected by the attacker, which returns the 5th byte in the MD5 hash of the bulletin board user with the userid of 3. By repeating this attack with different values for the second parameter of `substring`, the attacker can obtain the entire MD5 password hash of another user. The SQL injection policy described in Section 4 stops this attack.

- `SquirrelMail` *cross-site scripting* is present in version 1.2.10 of `SquirrelMail`, a popular web-based email client, e.g., `read_body.php` directly outputs values of user-controlled variables such as `mailbox` while generating HTML pages. The attack is stopped by the cross-site scripting policy in Section 4.

- `SquirrelMail` *command injection:* `SquirrelMail` (Version 1.4.0) constructs a command for encrypting email using the following statement in the function `gpg_encrypt` in the GPG plugin 1.1.

```
$command .= " -r $send_to_list 2>&1";
```

  The variable `send_to_list` should contain the recipient name in the "`To`" field, which is extracted using the `parseAddress` function of `Rfc822Header` object in `SquirrelMail`. However, due to a bug in this function, some malformed entries in the "`To`" field are returned without checking for proper email format. In particular, by entering "⟨recipient⟩; ⟨cmd⟩;" into this field, the attacker can execute any arbitrary command ⟨cmd⟩ with the privilege of the web server. By applying a policy that prohibits tainted shell meta-characters in the first argument to the `popen` function, this attack is stopped by our technique.

- `phpBB` *directory traversal:* A vulnerability exists in `phpBB`, which, when the gallery avatar feature is enabled, allows remote attackers to delete arbitrary files using directory traversal. This vulnerability can be exploited by a two-step attack. In the first step, the attacker saves the file name, which contains "`..`" characters, into the SQL database. In the second step, the file name is retrieved from the database and used in a command. To detect this attack, it is necessary to record taint information for data stored in the database, which is quite involved. We took a shortcut, and marked all data retrieved from the database as tainted. (Alternatively, we could have marked only those fields updated by the user as tainted.) This enabled the attack to be detected using the directory traversal policy.

- `phpxmlrpc/expat` *command injection:* `phpxmlrpc` is a package written in PHP to support the implementation of PHP clients and servers that communicate using the XML-RPC protocol. It uses the `expat` XML parser for processing XML. `phpxmlrpc` versions 1.0 and earlier have a remote command injection vulnerability. Our command injection policy stops exploitations of this vulnerability.

Bash CGI Application. `nph-test-cgi` is a CGI script that was included by default with Apache web server versions 1.0.5 and earlier. It prints out the values of the environment variables available to a CGI script. It uses the code `echo QUERY_STRING=$QUERY_STRING` to print the value of the query string sent to it. If the query string contains a "`*`" then `bash` will apply file name expansion to it, thus enabling an attacker to list any directory on the web server. This attack was stopped by a policy that restricted the use of tainted meta-characters in the argument to `shell_glob_filename`, which is the function used by `bash` for file name expansion. In terms of marking, the CGI interface defines the exact set of environment variables through which inputs are provided to a CGI application, and all these are marked as tainted.

6.2 False Positives

The policies described so far have been designed with the goal of avoiding false positives. We experimentally verified that false positives did not occur in our experiments involving the `wu-ftpd` server, the Apache

Server Programs	Workload	Orig. Response Time	Overhead
Apache-2.0.40	Webstone 30 clients downloading 5KB pages over 100Mbps network	0.036 sec/page	6%
wu-ftpd-2.6.0	Download a 12MB file 10 times.	11.5 sec	3%
postfix-1.1.12	Send one thousand 3KB emails	0.03 sec/mail	7%

Figure 6: Performance overheads of servers. For Apache server, performance is measured in terms of latency and throughput degradation. For other programs, it is measured in terms of overhead in client response time.

Program	Workload	Over-head(A)	Over-head(B)	Over-head(C)	Over-head(D)
bc-1.06	Find factorial of 600.	212%	68%	61%	**61%**
enscript-1.6.4	Convert a 5.5MB text file into a PS file.	660%	529%	63%	**58%**
bison-1.35	Parse a Bison file for C++ grammar.	134%	92%	79%	**78%**
gzip-1.3.3	Compress a 12 MB file.	228%	161%	110%	**106%**

Figure 7: Performance overheads of CPU-intensive programs. Performance is measured in terms of CPU time. Overheads in different columns correspond to: (A) No optimizations, (B) Use of local tag variable, (C) B + Use of 2-bit taint value, (D) C + Use of dependency analysis.

web server, and the two PHP applications, `phpBB` and `SquirrelMail`. For `wu-ftpd` and Apache, we enabled the control flow hijack policy, format string policy, directory traversal policy, and shell command injection policy. For the PHP applications, we additionally enabled the SQL injection policy and cross-site scripting policy for the PHP interpreter.

To evaluate the false positives for Apache, we used the transformed server as our lab's regular web server that accepted real-world HTTP requests from Internet for several hours. For the `wu-ftpd` server, we ran all the supported commands from a ftp client. To test `phpBB` and `SquirrelMail`, we went through all the menu items of these two Web applications, performed normal operations that a regular user might do, such as registering a user, posting a message, searching a message, managing address book, moving messages between different mail folders, and so on. No false positives were observed in these experiments.

6.3 False Negatives

False negatives can arise due to (a) overly permissive policies, (b) implicit information flows, and (c) use of untransformed libraries without adequate summarization functions.

We will discuss the policy refinement and implicit flows in Section 7. As for external libraries, the best approach is to transform them, so that the need for summarization can be eliminated. If this cannot be done, then our transformation will identify all the external functions that are used by an application, so that errors of omission can be avoided. However, if a summarization function is incorrect, then it can lead to false negatives, false positives, or both.

6.4 Performance

Figure 6 and 7 show the performance overheads, when the original and transformed programs were compiled using `gcc 3.2.2` with `-O2`, and ran on a 1.7GHz/512MB/Red Hat Linux 9.0 PC.

For server programs, the overhead of our approach is low. This is because they are I/O intensive, whereas our transformation adds overheads only to code that performs significant amount of data copying within the program, and/or other CPU-intensive operations. For CPU-intensive C programs, the overhead is between 61% to 106%, with an average of 76%.

6.4.1 Effect of Optimizations. The optimizations discussed in Section 3.3 have been very effective. We comment further in the context of CPU-intensive benchmarks.

- *Use of local taint variables* reduced the overheads by 42% to 144%. This is due to the reasons mentioned earlier: compilers such as `gcc` are very good in optimizing operations on local variables, but do a poor job on global arrays. Thus, by replacing global `tagmap` accesses with local tag variable accesses, significant performance improvement can be obtained.

  Most programs access local variables much more frequently than global variables. For instance, we found out (by instrumenting the code) that 99% of accesses made by `bc` are to local variables. A figure of 90% is not at all uncommon. As a result, the introduction of local tag variables leads to dramatic performance improvement for such programs. For programs that access global variables frequently, such as `gzip` that has 41% of its accesses going to global variables, the performance improvements are less striking.
- *`tagmap` optimizations* are particularly effective for

programs that operate mainly on integer data. This is because of the use of 2-bit taint tags, which avoids the need for bit-masking and shifts to access taint information. As a result we see significant overhead reduction in the range of 7% to 466%.

- *Intraprocedural analysis* and optimization further reduces the overhead by up to 5%. The gains are modest because `gcc` optimizations have already eliminated most local tag variables after the previous step.

When combined, these optimizations reduce the overhead by a factor of 2 to 5.

7 Discussion

7.1 Support for Implicit Information Flow

Implicit information flow occurs when the values of certain variables are related by virtue of program logic, even though there are no assignments between them. A classic example is given by the code snippet [25]:

```
x=x%2; y=0; if (x==1) y=1;
```

Even though there is no assignments involving `x` and `y`, their values are always the same. The need for tracking such implicit flows has long been recognized. [11] formalized implicit flows using a notion of *noninterference*. Several recent research efforts [18, 30, 20] have developed techniques based on this concept.

Noninterference is a very powerful property, and can capture even the least bit of correlation between sensitive data and other data. For instance, in the code:

```
if (x > 10000) error = true;
if (!error) { y = "/bin/ls"; execve(y); }
```

there is an implicit flow from `x` to `error`, and then to `y`. Hence, a policy that forbids tainted data to be used as an `execve` argument would be violated by this code. This example illustrates why non-interference may be too conservative (and hence lead to false positives) in our application. In the context of the kinds of attacks we are addressing, attackers usually need more control over the value of `y` than the minimal relationship that exists in the code above. Thus, it is more appropriate to track explicit flows. Nevertheless, there can be cases where substantial information flow takes place without assignments, e.g., in the following if-then-else, there is a direct flow of information from x to y on both branches, but our formulation of explicit information flow would only detect the flow in the else statement.

```
if (x == 0) y = 0; else y = x;
```

The goal of our approach is to support those implicit flows where the value of one variable *determines* the value of another variable. By using this criteria, we seek a balance between tracking necessary data value propagation and minimizing false positives. Currently, our implementation supports two forms of implicit flows that appear to be common in C programs.

- *Translation tables.* Decoding is sometimes implemented using a table look up, e.g.,

  ```
  y = translation_tab[x];
  ```

  where `translation_tab` is an array and `x` is a byte of input. In this case, the value of `x` determines the value of `y` although there is no direct assignment from `x` to `y`. To handle this case, we modify the basic transformation so that the result of an array access is marked as tainted whenever the subscript is tainted. This successfully handles the use of translation tables in the PHP interpreter.

- *Decoding using if-then-else/switch.* Sometimes, decoding is implemented using a statement of the form:

  ```
  if (x == '+') y = ' ';
  ```

  (Such code is often used for URL-decoding.) Clearly, the value of `y` can be determined by the value of `x`. More generally, `switch` statements could be used to translate between multiple characters. Our transformation handles them in the same way as a series of if-then-else statements. Specifically, consider an if-then-else statement of the form:

  `if (x ==` E`) {  ...  y =` E'`; ... }`

  If E and E' are constant-valued, then we add a tag update $tag(y) = tag(x)$ immediately before the assignment to y.

While our current technique seems to identify some of the common cases where implicit flows are significant, it is by no means comprehensive. Development of a more systematic approach that can provide some assurances about the kinds of implicit flows captured, while ensuring a low false positive rate, is a topic of future research.

7.2 Policy Refinement

Policy development effort is an important concern with any policy enforcement technique. In particular, there is a trade-off between policy precision and the level of effort required. If one is willing to tolerate false positives, policies that produce very few false negatives can be developed with modest effort. Alternatively, if false negatives can be tolerated, then false positives can be kept to a minimum with little effort. To contain both false positives and false negatives, more effort needs to be spent on policy development, taking application-specific or installation-specific characteristics.

The above remarks about policy-based techniques are generally applicable to our approach as well. For the format string attack, we used a policy that tended to err on the side of producing false positives, by disallowing all use of tainted format directives. However, it is conceivable that some applications may be prepared to receive a subset of format directives in untrusted inputs, and handle them correctly. In such cases, this application knowledge can be used by a system administrator to use a less restrictive policy, e.g., allowing the use of format directives other than `%n`. This should be done with care, or else it is possible to write policies that prevent the use of `%n`, but allow the use of variants such as `%5n` that have essentially the same effect. Alternatively, the policy may be relaxed to permit specific exceptions to the general rule that there be no format directives, e.g., the rule:

$$\begin{array}{l} \text{vfprintf}(fmt) \mid \\ \quad fmt\ \textbf{matches}\ any{*}\ (\text{Format})^{T} any{*}\ \ \&\& \\ \qquad (!(\text{fmt}\ \textbf{matches}\ \ \texttt{"[\^{}\%]*\%s[\^{}\%]*"})) \rightarrow reject() \end{array}$$

allows the use of a single `%s` format directive from untrusted sources, in addition to permitting format strings that contain untainted format directives.

The directory traversal policy also tends to err on the side of false positives, since it precludes all accesses outside the authorized top level directories (e.g. DocumentRoot and cgi-bin) of a web server if components of the file name being accessed are coming from untrusted sources. In devising this policy, we relied on application-specific knowledge, namely, the fact that web servers do not allow clients to access files outside the top level directories specified in the server configuration file. Another point to be noted about this policy is that variants of directory traversal attack that do not escape these top level directories, but simply attempt to fool per-directory access controls, are not addressed by our policy.

The control-flow hijack policy is already accurate enough to capture all attacks that use corruption of code pointers as the basis to alter the control-flow of programs, so we proceed to discuss the SQL injection policy. The policy shown in Figure 4 does not address attacks that inject only SQL keywords (e.g., the UNION operation) to alter the meaning of a query. This can be addressed by a policy based on tokenization. The idea is to perform a lexical analysis on the SQL query to break it up into tokens. SQL injection attacks are characterized by the fact that multiple tokens appear in the place of one, e.g., multiple keywords and meta-characters were provided by the attacker in the place of a simple string value in the attack examples discussed earlier in the paper. Thus, systematic protection from SQL injections can be obtained using a policy that prevents tainted strings from spanning multiple tokens. A similar approach is suggested in [24], although the conditions are not defined as precisely. Su et al [27] provide a formal characterization of SQL injection using a syntax analysis of SQL queries. The essential idea is to construct a parse tree for the SQL query, and to examine its subtrees. For any subtree whose root is tainted, all the nodes below that subtree should be tainted as well. In other words, tainted input cannot straddle different syntactic constructs. This is a further refinement over the characterization we suggest, where tainted input should not straddle different lexical entities.

Command injection attacks are similar to SQL injection attacks in many ways, and hence a tokenization-based policy may be a good choice for them as well. For this reason, we omit a detailed discussion of command injection policies. Nevertheless, it should be mentioned that there are some differences between SQL and command injection, e.g., shell syntax is much more complex than SQL syntax. Moreover, we may want to restrict the command names so that they are not tainted.

Note that tokenization is a lexical analysis task that is (almost invariably) implemented using regular expression based specifications. Thus, the above tokenization-based policy is amenable to expression using our policy language. One could argue that a regular expression to recognize tokens would be complex, and hence a policy may end up using a simpler approximation to tokenization. This discussion shows that the usual trade-off in policy based attack detection between accuracy and policy complexity continues in the case of taint-enhanced policies as well. Nevertheless, it should be noted that for a given policy development effort, taint-enhanced policies seem to be significantly more accurate than policies that do not incorporate any knowledge about taint.

Finally, we discuss the cross-site scripting attack. The policy discussed earlier does not address variations of the basic attack, e.g., attackers can evade this policy by injecting the malicious script code in "`onmouseover=malicious()`" or "`<img src="javascript:malicious()">`", which is not a block enclosed by the `script` tag. To detect these XSS variations, one has to understand the different HTML tag patterns in which a malicious script can be injected into dynamic HTML pages, and develop policies to prevent the use of such tainted patterns in HTML outputs.

In summary, although the example policies shown in Figure 4 were able to stop the attacks in our experiments, many of them need further improvement before they can stand up to skilled attackers that are knowledgeable about the policies being enforced. We outlined the ways to improve some of these policies, but a comprehensive solution to the policy development problem is not really the focus or contribution of this paper. Instead, our contribution is to show the feasibility and practicality of using fine-grained taint information in developing policy-based attack protection. The availability of fine-grained taint information makes our policies significantly more precise than traditional access-control policies. Moreover, our approach empowers system administrators and security professionals to update and refine these policies to improve protection, *without* having to wait for the patches of a newly discovered attack avenue.

8 Related Work

Memory Error Exploit Detection. Buffer overflows and related memory errors have received a lot of attention, and several efficient techniques have been developed to address them. Early approaches such as StackGuard [7] and ProPolice [9] focused on just a single class of attacks. Recently, more general techniques based on randomization have been developed, and they promise to defend against most memory error exploits [16, 2], However, due to the nature of the C language, these methods still cannot detect certain types of attacks, e.g., overflows from an array within a structure to an adjacent variable. Fine-grained taint analysis can capture these attacks whenever the corrupted data is used as an argument in a sensitive operation. (This is usually the case, since the goal of an attacker in corrupting that data was to perform a security-sensitive operation.) Although our overheads are generally higher than the techniques mentioned above, we believe that they are more than compensated by the increase in attack coverage.

Fine-Grained Taint Analysis. The key distinctions between our work and previous fine-grained taint analysis techniques of [22, 28, 5] were already discussed in the introduction, so we limit our discussion to the more technical points here. As mentioned earlier, [28, 5] rely on hardware support for taint-tracking. [22] is closer to our technique than these two techniques. It has an advantage over ours in that it can operate on arbitrary COTS binaries, whereas we require access to the C source code. This avoids problems such as hand-written assembly code. Their main drawback is performance: on the application Apache that they provide performance numbers on, their overheads are more than 100 times higher than ours. This is because (a) they rely on Valgrind, which in itself introduces more than 40 times overheads as compared to our technique, and (b) they are constrained by having to work on binary code, and without the benefit of static analyses and optimizations that have gone into our work. (Here, we are not only referring to our own analyses and optimizations, but also many of the optimizations implemented in the GCC compiler that we used to compile the transformed programs.)

There are several other technical differences between our work and that of [22]. For instance, they track 32-bits of taint information for each byte of data, whereas we use 2 bits. Another important difference is our support for implicit flows, which are not handled in [22].

Dynamic Taint Based Techniques for Detecting Attacks on Web Applications. Independently and in parallel to our work, which first appeared in [33], [23] and [24] have proposed the idea of using fine-grained taint analysis to detect injection attacks on web applications. The implementations of [23] and [24] are very similar, using hand-transformation of the PHP interpreter to track taint data. However, [24] provides a more detailed formulation and discussion of the problem, so we focus on this work here. They explain that these injection attacks are the result of *ad hoc* serialization of complex data such as SQL queries or shell commands, and develop a detection technique called *context-sensitive string evaluation (CSSE)*, which involves checking the use of tainted data in strings. Our work improves over theirs in several ways. First, by working at the level of the C language, we are able to handle many more applications: most server programs that are written in C, as well as programs written in interpreted languages such as PHP, bash and so on. Second, our formulation of the problem as taint-enhanced policy enforcement is more general, and can be applied to stealthy attacks such as those discussed in Section 2 that do not involve serialization problems; and to attacks involving arbitrary types of data rather than being limited to strings. Third, our approach relies on a simple transformation that is shown in Section 3, and implemented using 3.6KLOC of code, while their approach relies on manual transformation of a large piece of software that has over 300KLOC. Other technical contributions of our work include (a) the development of a simple policy language for concise specification of taint-enhanced polices, and (b) support for implicit flows that allow us to provide some support for character encodings and translations.

As discussed in Section 7, Su et al [27] describe a technique for detecting SQL injection attacks using syntax analysis. Their main focus is on providing a precise and formal characterization of SQL injection attacks. However, their implementation of taint tracking is not very reliable. In particular, they suggest a technique that avoids runtime operations for taint-tracking by "bracketing" each input string with two special symbols that surround untrusted input strings. Assuming that these brackets would be propagated together with input strings, checking for the presence of taint would reduce to checking for the presence of these special symbols. However, this assumption does not hold for programs that extract parts of their input and use them, e.g., a web application may remove non-alphanumeric characters from an input string and use them, and this process would likely discard the bracketing characters. In other cases, a web application may parse a user input into multiple fields, and use each field independently, once again causing the special symbols to be lost.

Manual Approaches for Correcting Input Validation Errors. Taint analysis targets vulnerabilities that arise due to missing or incorrect input validation code. One can manually review the code, and try to add all the necessary input validation checks. However, the notion of validity is determined by the manner in which the input is used. Thus, one has to trace forward in the program to identify all possible uses of an input in security sensitive operations, which is a very time-consuming and error-prone task. If we try to perform the validation check at the point of use, we face the problem that the notion of validity depends on the data source. For instance, it is perfectly reasonable for an SQL query to contain semicolons if these originated within the program text, but not so if it came from external input. Thus, we have to trace back from security-sensitive operations to identify how its arguments were constructed, once again having to manually examine large number of program paths. This leads to situations where validation checks are left out on some paths, and possibly duplicated on others. Moreover, the validation checks themselves are notoriously hard for programmers to code correctly, and have frequently been the source of vulnerabilities.

Information Flow. Information flow analysis has been researched for a long time [1, 10, 8, 18, 30, 20, 25]. Early research was focused on multi-level security, where fine-grained analysis was not deemed necessary [1]. More recent work has been focused on tracking information flow at variable level, and many interesting research results have been produced. While these techniques are promising for protecting privacy and integrity of sensitive data, as discussed in Section 2, the variable-level granularity is insufficient for detecting most attacks discussed in this paper.

Static Analysis. Static taint analysis techniques have been proposed by many for finding security vulnerabilities, including input validation errors in web applications [17, 14, 32], user/kernel pointer bugs [15], format string bugs [26], and bugs in placement of authorization hooks [34]. The main advantage of static analysis (as compared to runtime techniques) is that all potential vulnerabilities can be found statically, while its drawback is a relative lack of accuracy. In particular, these techniques typically detect *dependencies* rather than vulnerabilities. For instance, [17] will produce a warning whenever untrusted data is used in any manner in an SQL query. This may not be very useful if such a dependency is an integral part of application logic. To solve this problem, the concept of *endorsement* can be used to indicate "safe" dependencies. Typically, this is done by first performing appropriate validation checks on a piece of untrusted data, and then endorsing it to indicate that it is safe to use (i.e., no longer "tainted"). However, programmers are still responsible for determining what is "safe" — as discussed before, there is no easy way for them to do this.

An important difference between our work and static analysis is one of intended audience. Static analysis based tools are typically intended for use by developers, since they need detailed knowledge about program logic to determine where to introduce endorsements, and what validation checks need to be made before endorsement. In contrast, the audience for our tool is a system administrator or an outside security engineer that lacks detailed knowledge of application code.

Other Techniques. SQLrand [4] defeats SQL injection by randomizing the textual representation of SQL commands. A drawback of this approach, as compared to the technique presented in this paper, is that it requires manual changes to the program so that the program uses the modified representation for SQL commands generated by itself. Our approach was inspired by the effect achieved by SQLrand, namely, that of distinguishing commands generated by the application from those provided by untrusted users.

AMNESIA[12] is another interesting approach for detecting SQL injection attacks. It uses a static analysis of Java programs to compute a finite-state machine model that captures the lexical structure of SQL queries issued by a program. SQL injection attacks cause SQL queries issued by the program to deviate from this model, and hence detected. A key benefit of this approach is that by using static analysis, it can avoid runtime taint-tracking, and is hence much more efficient than our approach. Although this approach has been demonstrated to work well for SQL injections, the conservative nature of its static analysis and its inability to distinguish different sources of inputs can lead to a higher rate of false positives when applied to other types of attacks.

Perl has a taint mode [31] that tracks taint information at a coarse granularity – that of variables. In Perl, one has to explicitly untaint data before using it in a security sensitive context. This is usually done after performing appropriate validations. In our approach, due to the flexibility provided by our policy language, we have not faced a need for such explicit untainting. Nevertheless, if a user explicitly wants to trust some input, a primitive can be easily added to support this.

9 Conclusion

In this paper, we presented a unified approach that addresses a wide range of commonly reported attacks that exploit software implementation errors. Our approach is based on a fully automatic and efficient taint analysis technique that can track the flow of untrusted data through a program at the granularity of bytes. Through experiments, we showed that our technique can be applied to different types of applications written in multiple programming languages, and that it is effective in detecting attacks without producing false positives.

We believe that a number of software vulnerabilities arise due to the fact that security checks are interspersed throughout the program, and it is often difficult to check if the correct set of checks are being performed on every program path, especially in complex programs where the control flows through many, many functions. By decoupling policies from application logic, our approach can provide a higher degree of assurance on the correctness of policies. Moreover, the flexibility of our approach allows site administrators and third parties to quickly develop policies to prevent new classes of attacks, without having to wait for patches.

Acknowledgments

This research was supported in part by an ONR grant N000140110967 and NSF grants CNS-0208877 and CCR-0205376.

References

[1] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[2] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, August 2003.

[3] T. Bowen, D. Chee, M. Segal, R. Sekar, T. Shanbhag, and P. Uppuluri. Building survivable systems: An integrated approach based on intrusion detection and damage containment. In *DISCEX*, 2000.

[4] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In *International Conference on Applied Cryptography and Network Security (ACNS)*, pages 292–302, 2004.

[5] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *IEEE International Conference on Dependable Systems and Networks (DSN)*, 2005.

[6] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. FormatGuard: Automatic protection from printf format string vulnerabilities. In *USENIX Security Symposium*, 2001.

[7] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Automatic detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[8] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[9] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/main.html, June 2000.

[10] J. S. Fenton. Memoryless subsystems. *Computing Journal*, 17(2):143–147, May 1974.

[11] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, 1982.

[12] W. Halfond and A. Orso. AMNESIA: Analysis and monitoring for neutralizing SQL-injection. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2005.

[13] N. Hardy. The confused deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, October 1988.

[14] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *International World Wide Web Conference*, 2004.

[15] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, 2004.

[16] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *ACM Conference on Computer and Communication Security (CCS)*, 2003.

[17] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security Symposium*, 2005.

[18] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *IEEE Symposium on Security and Privacy*, pages 79–93, May 1994.

[19] S. McPeak, G. C. Necula, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for C program analysis and transformation. In *Conference on Compiler Construction*, 2002.

[20] A. C. Myers. JFlow: Practical mostly-static information flow control. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, Jan. 1999.

[21] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Workshop on Runtime Verification (RV)*, Boulder, Colorado, USA, July 2003.

[22] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security Symposium (NDSS)*, 2005.

[23] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *20th IFIP International Information Security Conference*, 2005.

[24] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection (RAID)*, 2005.

[25] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1), Jan. 2003.

[26] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, 2001.

[27] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 2006.

[28] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, Boston, MA, USA, 2004.

[29] P. Uppuluri and R. Sekar. Experiences with specification based intrusion detection. In *proceedings of the Recent Advances in Intrusion Detection conference*, October 2001.

[30] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[31] L. Wall, T. Christiansen, and R. Schwartz. *Programming Perl*. O'Reilly, 1996.

[32] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium*, 2006.

[33] W. Xu, S. Bhatkar, and R. Sekar. Practical dynamic taint analysis for countering input validation attacks on web applications. Technical Report SECLAB-05-04, Department of Computer Science, Stony Brook University, May 2005.

[34] X. Zhang, A. Edwards, and T. Jaeger. Using CQual for static analysis of authorization hook placement. In *USENIX Security Symposium*, 2002.

SANE: A Protection Architecture for Enterprise Networks

Martin Casado, Tal Garfinkel, Aditya Akella, Michael J. Freedman
Dan Boneh, Nick McKeown, Scott Shenker
{*casado,talg,mfreed,dabo,nickm*}*@cs.stanford.edu*
aditya@cs.cmu.edu, shenker@icsi.berkeley.edu

Abstract

Connectivity in today's enterprise networks is regulated by a combination of complex routing and bridging policies, along with various interdiction mechanisms such as ACLs, packet filters, and other middleboxes that attempt to retrofit access control onto an otherwise permissive network architecture. This leads to enterprise networks that are inflexible, fragile, and difficult to manage.

To address these limitations, we offer SANE, a protection architecture for enterprise networks. SANE defines a single *protection layer* that governs all connectivity within the enterprise. All routing and access control decisions are made by a logically-centralized server that grants access to services by handing out capabilities (encrypted source routes) according to declarative access control policies (e.g., "Alice can access http server *foo*"). Capabilities are enforced at each switch, which are simple and only minimally trusted. SANE offers strong attack resistance and containment in the face of compromise, yet is practical for everyday use. Our prototype implementation shows that SANE could be deployed in current networks with only a few modifications, and it can easily scale to networks of tens of thousands of nodes.

1 Introduction

The Internet architecture was born in a far more innocent era, when there was little need to consider how to defend against malicious attacks. Moreover, many of the Internet's primary design goals, such as universal connectivity and decentralized control, which were so critical to its success, are at odds with making it secure.

Worms, malware, and sophisticated attackers mean that security can no longer be ignored. This is particularly true for enterprise networks, where it is unacceptable to lose data, expose private information, or lose system availability. And so security measures have been retrofitted to enterprise networks via many mechanisms, including router ACLs, firewalls, NATs, and other middleboxes, along with complex link-layer technologies such as VLANs.

Despite years of experience and experimentation, these mechanisms are far from ideal. They require a significant amount of configuration and oversight [43], are often limited in the range of policies they can enforce [45], and produce networks that are complex [49] and brittle [50]. Moreover, even with these techniques, security within the enterprise remains notoriously poor. Worms routinely cause significant losses in productivity [9] and potential for data loss [29, 34]. Attacks resulting in theft of intellectual property and other sensitive information are similarly common [19].

The long and largely unsuccessful struggle to protect enterprise networks convinced us to start over with a clean slate, with security as a fundamental design goal. The result is our *Secure Architecture for the Networked Enterprise (SANE)*. The central design goals for our architecture are as follows:

- *Allow natural policies that are simple yet powerful.* We seek an architecture that supports natural policies that are independent of the topology and the equipment used, e.g., "Allow everyone in group sales to connect to the http server hosting documentation." This is in contrast to policies today that are typically expressed in terms of topology-dependent ACLs in firewalls. Through high-level policies, our goal is to provide access control that is restrictive (i.e., provides least privilege access to resources), yet flexible, so the network does not become unusable.

- *Enforcement should be at the link layer, to prevent lower layers from undermining it.* In contrast, it is common in today's networks for network-layer access controls (e.g., ACLs in firewalls) to be undermined by more permissive connectivity at the link layer (e.g., Ethernet and VLANs).

- *Hide information about topology and services from those without permission to see them.* Once an attacker has compromised an end host, the usual next step is to map out the network's topology—to identify firewalls, critical servers, and the location of end hosts—and to identify end hosts and services that can be compromised. Our goal is to hide all such information to embrace the principle of least knowledge.

- *Have only one trusted component.* Today's networks trust multiple components, such as firewalls, switches, routers, DNS, and authentication services (e.g., Kerberos, AD, and Radius). The compromise of any one component can wreak havoc on the entire enterprise. Our goal is to rely on a central (yet potentially replicated) trusted entity where all policy is centrally defined and executed.

SANE achieves these goals by providing a single protection layer that resides between the Ethernet and IP layer, similar to the place that VLANs occupy. All connectivity is granted by handing out *capabilities*. A capability is an encrypted source route between any two communicating end points.

Source routes are constructed by a logically-centralized Domain Controller (DC) with a complete view of the network topology. By granting access using a global vantage point, the DC can implement policies in a topology-independent manner. This is in contrast to today's networks: the rules in firewalls and other middleboxes have implicit dependencies on topology, which become more complex as the network and policies grow (e.g. VLAN tagging and firewall rules) [14, 47].

By default, hosts can only route to the DC. Users must first authenticate themselves with the DC before they can request a capability to access services and end hosts. Access control policies are specified in terms of services and principals, e.g., "users in group martins-friends can access martin's streaming-audio server".

At first glance, our approach may seem draconian: All communication requires the permission of a central administrator. In practice, the administrator is free to implement a wide variety of policies that vary from strict to relaxed and differ among users and services. The key here is that SANE allows the easy implementation and enforcement of a simply expressed rule.

Our approach might also seem dependent on a single point-of-failure (the DC) and not able to route traffic around failures (because of static source routes). However, as we will argue, we can use standard replication techniques, such as multiple DCs and redundant source routes, to make the network reliable and quick to recover from failures.

The remainder of the paper is organized as follows. In Section 2, we further argue why current security mechanisms for the enterprise are insufficient and why the SANE approach is feasible. Section 3 presents a detailed design of SANE. We will see that by delegating access control and routing to a central controller, we can reduce the complexity of the forwarding elements (switches) and the degree to which we must trust them. We also show how a specific implementation of SANE could be deployed in current networks with only a few modifications (even though SANE is a radical departure from traditional network design). Section 4 covers SANE's resistance to a strong attack model. In Section 5, we present and evaluate a prototype software implementation of SANE, and Section 6 demonstrates that SANE can easily scale to networks of tens of thousands of nodes and does not significantly impact user-perceived latency. We present related work in Section 7 and conclude in Section 8.

2 What's Wrong with Existing Techniques?

Complexity of Mechanism. A typical enterprise network today uses several mechanisms simultaneously to protect its network: VLANs, ACLs, firewalls, NATs, and so on. The security policy is distributed among the boxes that implement these mechanisms, making it difficult to correctly implement an enterprise-wide security policy. Configuration is complex (for example, routing protocols often require thousands of lines of policy configuration [50]), making the security fragile. Furthermore, the configuration is often dependent on network topology, and is based on addresses and physical ports, rather than on authenticated end-points. When the topology changes or hosts move, the configuration frequently breaks, requires careful repair [50], and possibly undermines its security policies.

A common response is to put all security policy in one box and at a choke-point in the network, for example, in a firewall at the network's entry and exit point. If an attacker makes it through the firewall, they have unfettered access to the whole network.

Another way to address this complexity is to enforce protection on the end host via distributed firewalls [14]. While reasonable, this has the down-side of placing all trust in the end hosts. End host firewalls can be disabled or bypassed, leaving the network unprotected, and they offer no containment of malicious infrastructure, e.g., a compromised NIDS [8].

Our new architecture allows simple high-level policies to be expressed centrally. Policies are enforced by a single fine-grain mechanism within the network.

Proliferation of Trust. Today's networks provide a fertile environment for the skilled attacker. Switches and routers must correctly export link state, calculate routes, and perform filtering; yet over time, these mechanisms have become more complex, with new vulnerabilities discovered at an alarming rate [8, 10, 7, 11]. If compromised, an attacker can often take down the network [32, 48] or redirect traffic to permit eavesdropping, traffic analysis, and man-in-the-middle attacks.

Our new architecture replaces all these mechanisms with simple, minimally-trusted forwarding elements, reducing the number of trusted (and configured) components to just one centrally-managed controller. Our goal is to minimize the trusted computing base.

Proliferation of Information. A further resource for an attacker is the proliferation of information on the network layout of today's enterprises. This knowledge is valuable for helping to identify sensitive servers, firewalls, and IDS systems, which can be exploited for compromise or denial of service. Topology information is easy to gather: switches and routers keep track of the network topology (e.g., the OSPF topology database) and broadcast it periodically in plain-text. Likewise, host enumeration (e.g., ping and ARP scans), port scanning, traceroutes, and SNMP can easily reveal the existence of, and the route to, hosts. Today it is common for network operators to filter ICMP and change default SNMP passphrases to limit the amount of information available to an intruder.

Our new architecture hides both the network structure, as well as the location of critical services and hosts, from all unauthorized network entities. Minimal information is made available as needed for correct function and for fault diagnosis.

2.1 Threat Environment

SANE seeks to provide protection robust enough for demanding threat environments—government and military networks, financial institutions, or demanding business settings—yet flexible enough for everyday use. We assume a robust threat environment with both *insider* (authenticated users or switches) and *outsider* threats (e.g., an unauthenticated attacker plugging into a network jack). This attacker may be sophisticated, capable of compromising infrastructure components and exploiting protocol weaknesses. Consequently, we assume attacks can originate from any network element, such as end hosts, switches, or firewalls.

SANE prevents outsiders from originating any traffic except to the DC, while preventing malicious end hosts from either sending traffic anywhere that has not been explicitly authorized, or, if authorized, subjecting the network to a denial-of-service attack which cannot be subsequently disabled.

SANE makes a best effort attempt to maintain availability in the face of malicious switches; however, we do not attempt to achieve full network-layer Byzantine fault tolerance [38]. In a normal SANE network, little can be done in the face of a malicious DC, however, we discuss strategies for dealing with this and other threats in §4.

2.2 What's Special about the Enterprise?

We can exploit several properties of enterprise networks to make them more secure. First, enterprise networks are often carefully engineered and centrally administered, making it practical (and desirable) to implement policies in a central location.[1]

Second, most machines in enterprise networks are clients that typically contact a predictable handful of local services (e.g., mail servers, printers, file servers, source repositories, HTTP proxies, or ssh gateways). Therefore, we can grant relatively little privilege to clients using simple declarative access control policies; in our system we adopt a policy interface similar to that of a distributed file system.

Third, in an enterprise network, we can assume that hosts and principals are authenticated; this is already common today, given widely deployed directory services such as LDAP and Active Directories. This allows us to express policies in terms of meaningful entities, such as hosts and users, instead of weakly bound end-point identifiers such as IP and MAC addresses.

Finally, enterprise networks—when compared to the Internet at large—can quickly adopt a new protection architecture. "Fork-lift" upgrades of entire networks are not uncommon, and new networks are regularly built from scratch. Further, there is a significant willingness to adopt new security technologies due to the high cost of security failures.

3 System Architecture

SANE ensures that network security policies are enforced during all end host communication at the link layer, as shown in Figure 1. This section describes two versions of the SANE architecture. First, we present a clean-slate approach, in which every network component is modified to support SANE. Later, we describe a version of SANE that can inter-operate with unmodified end hosts running standard IP stacks.

3.1 Domain Controller

The Domain Controller (DC) is the central component of a SANE network. It is responsible for authenticating

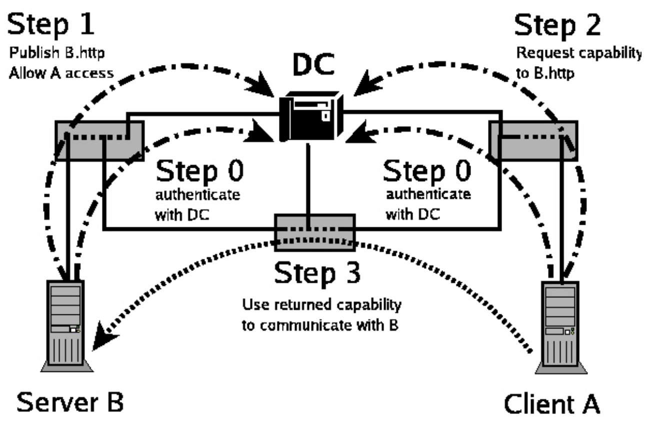


Figure 1: The SANE Service Model: By default, SANE only allows hosts to communicate with the Domain Controller (DC). To obtain further connectivity they must take the following steps: (0) Principals authenticate to the DC and establish a secure channel for future communication. (1) Server *B publishes* service under a unique name B.http in the Network Service Directory. (2) For a client *A* to get permission to access B.http, it obtains a *capability* for the service. (3) Client *A* can now communicate with server by prepending the returned capability to each packet.

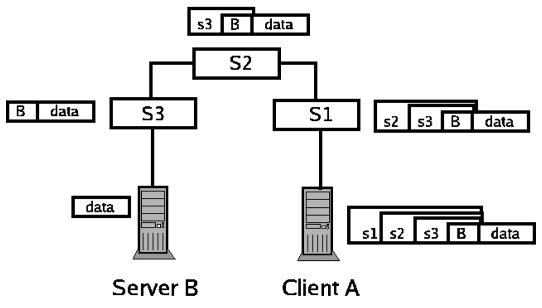


Figure 2: Packets forwarded from client A to server B across multiple switches using a source-routed capability. Each layer contains the next-hop information, encrypted to the associated switch's symmetric key. The capability is passed to A by the DC (not shown) and can be re-used to send packets to B until it expires.

Ethernet	SANE header	IP header	data

Figure 3: SANE operates at the same layer as VLAN. All packets on the network must carry a SANE header at the *isolation layer*, which strictly defines the path that packet is allowed to take.

users and hosts, advertising services that are available, and deciding who can connect to these services. It allows hosts to communicate by handing out capabilities (encrypted source routes). As we will see in Section 3.5, because the network depends on it, the DC will typically be physically replicated (described in Section 3.5).

The DC performs three main functions:

1. **Authentication Service**: This service authenticates principals (e.g., users, hosts) and switches. It maintains a symmetric key with each for secure communication.[2]

2. **Network Service Directory (NSD)**: The NSD replaces DNS. When a principal wants access to a service, it first looks up the service in the NSD (services are published by servers using a unique name). The NSD checks for permissions—it maintains an access control list (ACL) for each service—and then returns a *capability*. The ACL is declared in terms of system principals (users, groups), mimicking the controls in a file system.

3. **Protection Layer Controller**: This component controls all connectivity in a SANE network by generating (and revoking) *capabilities*. A capability is a switch-level *source route* from the client to a server, as shown in Figure 2. Capabilities are encrypted in layers (i.e., onion routes [23]) both to prove that they originated from the DC and to hide topology. Capabilities are included in a SANE header in all data packets. The SANE header goes between the Ethernet and IP headers, similar to the location VLANs occupy (Figure 3).

   The controller keeps a complete view of the network topology so that it can compute routes. The topology is constructed on the basis of link-state updates generated by authenticated switches. Capabilities are created using the symmetric keys (to switches and hosts) established by the authentication service.

   The controller will adapt the network when things go wrong (maliciously or otherwise). For example, if a switch floods the DC with control traffic (e.g. link-state updates), it will simply eliminate the switch from the network by instructing its immediate neighbor switches to drop all traffic from that switch. It will issue new capabilities so that ongoing communications can start using the new topology.

All packet forwarding is done by switches, which can be thought of as simplified Ethernet switches. Switches forward packets along the encrypted source route carried in each packet. They also send link-state updates to the DC so that it knows the network topology.

Note that, in a SANE network, IP continues to provide wide-area connectivity as well as a common framing format to support the use of unmodified end hosts. Yet within a SANE enterprise, IP addresses are not used for identification, location, nor routing.

3.2 Network Service Directory

The NSD maintains a hierarchy of directories and services; each directory and service has an access control list specifying which users or groups can view, access, and publish services, as well as who can modify the ACLs. This design is similar to that deployed in distributed file systems such as AFS [25].

As an example usage scenario, suppose `martin` wants to share his MP3's with his friends `aditya`, `mike`, and `tal` in the high performance networking group. He sets up a streaming audio server on his machine `bongo`, which has a directory `stanford.hpn.martin.friends` with ACLs already set to allow his friends to list and acquire services. He publishes his service by adding the command

```
sane --publish stanford.martin.ambient:31337
```

to his audio server's startup script, and, correspondingly, adds the command

```
sane --remove stanford.martin.ambient
```

to its shutdown script. When his streaming audio server comes on line, it publishes itself in the NSD as `ambient`. When `tal` accesses this service, he simply directs his MP3 player to the name `stanford.martin.ambient` The NSD resolves the name (similar to DNS), has the DC issue a capability, and returns this capability, which `tal`'s host then uses to access the audio server on `bongo`.

There is nothing unusual about SANE's approach to access control. One could envision replacing or combining SANE's simple access control system with a more sophisticated trust-management system [15], in order to allow for delegation, for example. For most purposes, however, we believe that our current model provides a simple yet expressive method of controlling access to services.

3.3 Protection Layer

All packets in a SANE network contain a SANE header located between the Ethernet and IP headers. In Figure 4, we show the packet types supported in SANE, as well as their intended use (further elaborated below).

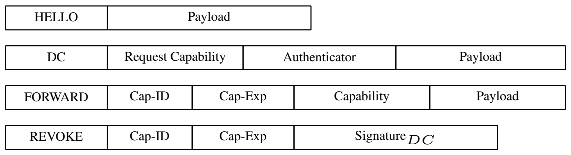


Figure 4: Packet types in a SANE network: `HELLO` packets are used for immediate neighbor discovery and thus are never forwarded. `DC` packets are used by end hosts and switches to communicate with the DC; they are forwarded by switches to the DC along a default route. `FORWARD` packets are used for most host-to-host data transmissions; they include an encrypted source route (capability) which tells switches where to forward the packet. Finally, `REVOKE` packets revoke a capability before its normal expiration; they are forwarded back along a capability's forward route.

Communicating with the DC. SANE establishes default connectivity to the DC by building a minimum spanning tree (MST), with the DC as the root of the tree. This is done using a standard distance vector approach nearly identical to that used in Ethernet switches [1], with each switch sending `HELLO` messages to its neighbor, indicating its distance from the root. The MST algorithm has the property that no switch learns the network topology nor is the topology reproducible from packet traces.

The spanning tree is only used to establish default routes for forwarding packets to the DC. We also need a mechanism for the DC to communicate back with switches so as to establish symmetric keys, required both for authentication and for generating and decoding capabilities. Note that the DC can initially only communicate with its immediate neighbors, since it does not know the full topology.

The DC first establishes shared keys with its direct neighbors, and it receives link-state updates from them. It then iteratively contacts switches at increasing distances (hop-counts), until it has established shared keys with all switches to obtain a map of the full topology.[3] To contact a switch multiple hops away, the DC must first generate a capability given the topology information collected thus far. Once established, keys provide confidentiality, integrity, and replay defense for all subsequent traffic with the DC via an *authenticator* header, much like IPsec's ESP header.

All capability requests and link state updates—packets of type `DC`—are sent along the MST. As packets traverse the MST, the switches construct a *request capability*[4] by generating an encrypted onion at each hop containing the previous and next hop, encrypted under the switch's own key. The DC uses the request capabilities to communicate back to each sender. Because these capabilities encode the path, the DC can use them to determine the location of misbehaving senders.

Point-to-Point Communication. Hosts communicate using capabilities provided by the DC. This traffic is sent using `FORWARD` packets which carry the capability. On receipt of a packet, switches first check that the capability is valid, that it has not expired and that it has not been revoked (discussed later).

Before discussing how capabilities are constructed, we must differentiate between long-lived names and ephemeral connection identifiers. Names are known to the service directory for published services and their access control lists. Identifiers enable end hosts to demultiplex packets as belonging to either particular connections with other end hosts or to capability requests with the DC, much like transport-level port numbers in TCP or UDP. (They are denoted as `client-ID` and `server-ID` below.) So, much like in traditional networks à la DNS names and IP addresses, users use SANE names to identify end-points, while the network software and hardware uses connection identifiers to identify unique services.

The DC constructs capabilities using three pieces of information: the client's name and location (given in the capability request), the service's location (stored in the service directory), and the path between these two end-points (as calculated from the network topology and any service policies).

Each layer in the capability is calculated recursively, working backward from the receiver, using the shared key established between the DC and the corresponding switches.

1. Initialize:
   CAPABILITY $\leftarrow$ $E_{K_{\mathrm{server-name}}}$ (client-name, client-ID, server-ID, last-hop)
2. Recurse: For each node on the path, starting from the last node, do:
   CAPABILITY $\leftarrow E_{K_{\mathrm{switch-name}}}$ (switch-name, next-hop, prev-hop, CAPABILITY)
3. Finalize:
   CAPABILITY $\leftarrow E_{K_{\mathrm{client-name}}}$ (client-name, client-ID, first-hop, CAPABILITY), IV

Where, $E_k(m)$ denotes the encryption of message m under the symmetric key k. Encryption is implemented using a block cipher (such as AES) and a message authentication code (MAC) to provide both confidentiality and integrity.

All capabilities have a globally unique ID `Cap-ID` for revocation, as well as an expiration time, on the order of a few minutes, after which a client must request a new capability. This requires that clocks are only loosely synchronized to within a few seconds. Expiration times may vary by service, user, host, etc.

The MAC computation for each layer includes the `Cap-ID` as well as the expiration time, so they cannot be tampered with by the sender or en-route. The initialization vector (IV) provided in the outer layer of capabilities is the encryption randomization value used for all layers. It prevents an eavesdropper from linking capabilities between the same two end-points.[5]

Revoking Access. The DC can *revoke* a capability to immediately stop a misbehaving sender for misusing a capability. A victim first sends a revocation request, which consists of the final layer of the offending capability, to the DC. The DC verifies that the requester is on the capability's path, and it returns a signed packet of type `REVOKE`.

The requester then pushes the revocation request to the upstream switch from which the misbehaving capability was forwarded. The packet travels hop-by-hop on the reverse path of the offending capability. On-path switches verify the DC's digital signature, add the revoked `Cap-ID` to a local revocation list, and compare it with the `Cap-ID` of each incoming packet. If a match is found, the switch drops the incoming packet and forwards the revocation to the previous hop. Because such revocation packets are not on the data path, we believe that the overhead of signature verification is acceptable.

A revocation is only useful during the lifetime of its corresponding capability and therefore carries the same expiration time. Once a revocation expires, it is purged from the switch. We discuss protection against revocation state exhaustion in section 4.1.

3.4 Interoperability

Discussion thus far has assumed a clean-slate redesign of all components in the network. In this section, we describe how a SANE network can be used by unmodified end-hosts with the addition of two components: *translation proxies* for mapping IP events to SANE events and *gateways* to provide wide-area connectivity.

Translation Proxies. These proxies are used as the first hop for all unmodified end hosts. Their primary function is to translate between IP naming events and SANE events. For example, they map DNS name queries to DC service lookups and DC lookup replies to DNS replies. When the DC returns a capability, the proxy will cache it and add it to the appropriate outgoing packets from the host. Conversely, the proxy will remove capabilities from packets sent to the host.

In addition to DNS, there are a number of service discovery protocols used in today's enterprise networks, such as SLP [44], DNS SD [4], and uPNP [6]. In order to be fully backwards-compatible, SANE translation proxies must be able to map all service lookups and requests to DC service queries and handle the responses.

Gateways. Gateways provide similar functionality to perimeter NATs. They are positioned on the perimeter of a SANE network and provide connectivity to the wide area. For outgoing packets, they cache the capability and generate a mapping from the IP packet header (e.g., IP/port 4-tuple) to the associated capability. All incoming packets are checked against this mapping and, if one exists, the appropriate capability is appended and the packet is forwarded.

Broadcast. Unfortunately, some discovery protocols, such as uPNP, perform service discovery by broadcasting lookup requests to all hosts on the LAN. Allowing this without intervention would be a violation of least privilege. To safely support broadcast service discovery within SANE, all packets sent to the link-layer broadcast address are forwarded to the DC, which verifies that they strictly conform to the protocol spec. The DC then reissues the request to all end hosts on the network, collects the replies and returns the response to the sender. Putting the DC on the path allows it to cache services for subsequent requests, thus having the additional benefit of limiting the amount of broadcast traffic. Designing SANE to efficiently support broadcast and multicast remains part of our future work.

Service Publication. Within SANE, services can be published with the DC in any number of ways: translating existing service publication events (as described above), via a command line tool, offering a web interface, or in the case of IP, hooking into the `bind` call on the local host à la SOCKS [30].

3.5 Fault Tolerance

Replicating the Domain Controller. The DC is logically centralized, but most likely physically replicated so as to be scalable and fault tolerant. Switches connect to multiple DCs through multiple spanning trees, one rooted at each DC. To do this, switches authenticate and send their neighbor lists to each DC separately. Topology consistency between DCs is not required as each DC grants routes independently. Hosts randomly choose a DC to send requests so as to distribute load.

Network level-policy, user declared access policy and the service directory must maintain consistency among multiple DCs. If the DCs all belong to the same enterprise–and hence trust each other–service advertisements and access control policy can be replicated between DCs using existing methods for ensuring distributed consistency. (We will consider the case where DCs do not trust each other in the next section.)

Recovering from Network Failure. In SANE, it is the end host's responsibility to determine network failure. This is because direct communication from switches to end hosts violates least privilege and creates new avenues for DoS. SANE-aware end hosts send periodic probes or keep-alive messages to detect failures and request fresh capabilities.

When a link fails, a DC will be flooded with requests for new capabilities. We performed a feasibility study (in Section 6), to see if this would be a problem in practice, and found that even in the worst-case when all flows are affected, the requests would not overwhelm a single DC.

So that clients can adapt quickly, a DC may issue multiple (edge-disjoint, where possible) capabilities to clients. In the event of a link failure, a client simply uses another capability. This works well if the topology is rich enough for there to be edge-disjoint paths. Today's enterprise networks are not usually richly interconnected, in part because additional links and paths make security more complicated and easier to undermine. However, this is no longer true with SANE—each additional switch and link improves resilience. With just two or three alternate routes we can expect a high degree of fault tolerance [27]. With multiple paths, an end host can set aggressive time-outs to detect link failures (unlike in IP networks, where convergence times can be high).

3.6 Additional Features

This section discusses some additional considerations of a SANE network, including its support for middleboxes, mobility, and support for logging.

Middleboxes and Proxies. In today's networks, proxies are usually placed at choke-points, to make sure traffic will pass through them. With SANE, a proxy can be placed anywhere; the DC can make sure the proxy is on the path between a client and a server. This can lead to powerful application-level security policies that far outreach port-level filtering.

At the very least, lightweight proxies can validate that communicating end-points are adhering to security policy. Proxies can also enforce service- or user-specific policies or perform transformations on a per-packet basis. These could be specified by the capability. Proxies might scan for viruses and apply vulnerability-specific filters, log application-level transactions, find information leaks, and shape traffic.

Mobility. Client mobility within the LAN is transparent to servers, because the service is unaware of (and so independent of) the underlying topology. When a client changes its position—e.g., moves to a different wireless access point—it refreshes its capabilities and passes new return routes to the servers it is accessing. If a client moves locations, it should revoke its current set of outstanding capabilities. Otherwise, much like today, a new machine plugged into the same access point could access traffic sent to the client after it has left.

Server mobility is handled in the same manner as adapting to link failures. If a server changes location, clients will detect that packets are not getting through and request a new set of capabilities. Once the server has updated its service in the directory, all (re)issued capabilities will contain the correct path.

Anti-mobility. SANE also trivially anti-mobility. That is, SANE can *prevent* hosts and switches from moving on the network by disallowing access if they do. As the DC knows the exact location of all senders given request capabilities, it can be configured to only service hosts if they are connected at particular physical locations. This is useful for regulatory compliance, such as 911 restrictions on movement for VoIP-enabled devices. More generally, it allows a strong "lock-down" of network entities to enforce strong policies in the highest-security networks. For example, it can be used to disallow all network access to rogue PCs.

Centralized Logging. The DC, as the broker for all communications, is in an ideal position for network-wide connection logging. This could be very useful for forensics. Request routes protect against source spoofing on connection setup, providing a path back to the connecting port in the network. Further, compulsory authentication matches each connection request to an actual user.

4 Attack Resistance

SANE eliminates many of the vulnerabilities present in today's networks through centralization of control, simple declarative security policies and low-level enforcement of encrypted source routes. In this section, we enumerate the main ways that SANE resists attack.

- **Access-control lists:** The NSD uses ACLs for *directories*, preventing attackers from enumerating all services in the system—an example of the principle of least knowledge—which in turn prevents the discovery of particular applications for which compromises are known. The NSD controls access to *services* to enforce protection at the link layer through DC-generated capabilities—supporting the principle of least privilege—which stops attackers from compromising applications, even if they are discovered.

- **Encrypted, authenticated source-routes and link-state updates:** These prevent an attacker from learning the topology or from enumerating hosts and performing port scans, further examples of the principle of least knowledge.[6] SANE's source routes prevent hosts from spoofing requests either to the DC on the control path or to other end hosts on the data path. We discuss these protections further in Section 4.1.

- **Authenticated network components:** The authentication mechanism prevents unauthenticated switches from joining a SANE network, thwarting a variety of topology attacks. Every switch enforces capabilities providing defence in depth. Authenticated switches cannot lie about their connectivity to create arbitrary links, nor can they use the same authenticated public key to join the network using different physical switches. Finally, well-known spanning-tree or routing attacks [32, 48] are impossible, given the DC's central role. We discuss these issues further in section 4.2.

SANE attempts to degrade gracefully in the face of more sophisticated attacks. Next, we examine several major classes of attacks.

4.1 Resource Exhaustion

Flooding. As discussed in section 3.3, flooding attacks are handled through revocation. However, misbehaving switches or hosts may also attempt to attack the network's control path by flooding the DC with requests. Thus, we rate-limit requests for capabilities to the DC. If a switch or end host violates the rate limit, the DC tells its neighbors to disconnect it from the network.

Revocation state exhaustion. SANE switches must keep a list of revoked capabilities. This list might fill, for example, if it is maintained in a small CAM. An attacker could hoard capabilities, then cause all of them to be revoked simultaneously. SANE uses two mechanisms to protect against this attack: (1) If its revocation list fills, a switch simply generates a new key; this invalidates all existing capabilities that pass through it. It clears its revocation list, and passes the new key to the DC. (2) The DC tracks the number of revocations issued per sender. When this number crosses a predefined threshold, the sender is removed from the service's ACLs.

If a switch uses a sender's capability to flood a receiver, thus eliciting a revocation, the sender can use a different capability (if it has one) to avoid the misbehaving switch. This occurs naturally because the client treats revocation—which results in an inability to get packets

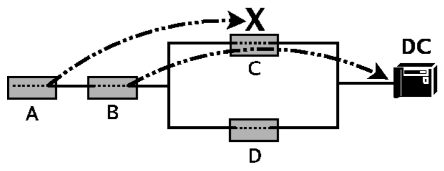


Figure 5: Attacker **C** can deny service to **A** by selectively dropping **A**'s packets, yet letting the packets of its parent (**B**) through. As a result, **A** cannot communicate with the DC, even though a alternate path exists through **D**.

through—as a link failure, and it will try using a different capability instead. While well-behaved senders may have to use or request alternate capabilities, their performance degradation is only temporary, provided that there exists sufficient link redundancy to route around misbehaving switches. Therefore, using this approach, SANE networks can quickly converge to a state where attackers hold no valid capabilities and cannot obtain new ones.

4.2 Tolerating Malicious Switches

By design, SANE switches have minimal functionality—much of which is likely to be placed in hardware—making remote compromise unlikely. Furthermore, each switch requires an authenticated public key, preventing rogue switches from joining the network. However, other avenues of attack, such as hardware tampering or supply-chain attacks, may allow an adversary to introduce a malicious switch. For completeness, therefore, we consider defenses against malicious switches attempting to sabotage network operation, even though the following attacks are feasible only in the most extreme threat environments.

Sabotaging MST Discovery. By falsely advertising a smaller distance to the DC during MST construction, a switch can cause additional DC traffic to be routed through it. Nominally, this practice can create a path inefficiency.

More seriously, a switch can attract traffic, then start dropping packets. This practice will result in degraded throughput, unless the drop rate increases to a point at which the misbehaving switch is declared failed and a new MST is constructed.

In a more subtle attack, a malicious switch can selectively allow packets from its neighbors, yet drop all other traffic. An example of this attack is depicted in Figure 5: Node C only drops packets from node A. Thus, B does not change its forwarding path to the DC, as C appears to be functioning normal from its view. As a result, A cannot communicate with the DC, even though an alternate path exists through D. Note that this attack, at the MST discovery phase, precludes our normal solution for routing around failures—namely, using node-disjoint paths whenever possible—as node A has never registered with the DC in the first place.

From a high level, we can protect against this selective attack by hiding the identities of senders from switches en-route. Admittedly, it is unlikely that we can prevent *all* such information leakage through the various side-channels that naturally exist in a real system, e.g., due to careful packet inspection and flow analysis. Some methods to confound such attacks include (1) hiding easily recognizable sender-IDs from packet headers,[7] (2) padding all response capabilities to the same length to hide path length, and (3) randomizing periodic messages to the DC to hide a node's scheduled timings.

Using these safeguards, if a switch drops almost all packets, its immediate neighbors will construct a new MST that excludes it. If it only occasionally drops packets, the rate of MST discovery is temporarily degraded, but downstream switches will eventually register with the DC.

Bad Link-State Advertisements. Malicious switches can try to attract traffic by falsifying connectivity information in link-state updates. A simple safeguard against such attacks is for the DC to only add non-leaf edges to its network map when both switches at either end have advertised the link.

This safeguard does not prevent colluding nodes from falsely advertising a link between themselves. Unfortunately, such collusion cannot be externally verified. Notice that such collusion can only result in a temporary denial-of-service attack when capabilities containing a false link are issued: When end hosts are unable to route over a false link, they immediately request a fresh capability. Additionally, the isolation properties of the network are still preserved.

Note that SANE's requirement for switches to initially authenticate themselves with the DC prevents Sybil attacks, normally associated with open identity-free networks [21].

4.3 Tolerating a Malicious DC

Domain controllers are highly trusted entities in a SANE network. This can create a single point-of-failure from a security standpoint, since the compromise of any one DC yields total control to an attacker.

To prevent such a take-over, one can distribute trust among DCs using threshold cryptography. While the full details are beyond the scope of this paper, we sketch the basic approach. We split the DCs' secret key across a few servers (say $n < 6$), such that two of them are needed to generate a capability. The sender then communicates with 2-out-of-n DCs to obtain the capability. Thus, an attacker gains no additional access by compromising a single DC.[8] To prevent a single malicious DC from revoking arbitrary capabilities or, even worse, completely disconnecting a switch or end host, the revocation mechanism (section 3.3) must also be extended to use asymmetric threshold cryptography [20].

Given such replicated function, access control policy and service registration must be done independently with each DC by the end host, using standard approaches for consistency such as two-phase commit. When a new DC comes online or when a DC re-establishes communication after a network partition, it must have some means of re-syncing with the other DCs. This can be achieved via standard Byzantine agreement protocols [18].

5 Implementation

This section describes our prototype implementation of a SANE network. Our implementation consists of a DC, switches, and IP proxies. It does not support multiple DCs, there is no support for tolerating malicious switches nor were any of the end-hosts instrumented to issue revocations.

All development was done in C++ using the Virtual Network System (VNS) [17]. VNS provides the ability to run processes within user-specified topologies, allowing us to test multiple varied and complex network topologies while interfacing with other hosts on the network. Working outside the kernel provided us with a flexible development, debug, and execution environment.

The network was in operational use within our group LAN—interconnecting seven physical hosts on 100 Mb Ethernet used daily as workstations—for one month. The only modification needed for workstations was to reduce the maximum transmission unit (MTU) size to 1300 bytes in order to provide room for SANE headers.

5.1 IP Proxies and SANE Switches

To support unmodified end hosts on our prototype network, we developed proxy elements which are positioned between hosts and the first hop switches. Our proxies use ARP cache poisoning to redirect all traffic from the end hosts. Capabilities for each flow are cached at the corresponding proxies, which insert them into packets from the end host and remove them from packets to the end host.

Our switch implementation supports automatic neighbor discovery, MST construction, link-state updates and packet forwarding. Switches exchange `HELLO` messages every 15 seconds with their neighbors. Whenever switches detects network failures, they reconfigure their MST and update the DC's network map.

The only dynamic state maintained on each switch is a hash table of capability revocations, containing the `Cap-ID`s and their associated expiration times.

We use OCB-AES [42] for capability construction and decryption with 128-bit keys. OCB provides both confidentiality and data integrity using a single pass over the data, while generating ciphertext that is exactly only 8 bytes longer than the input plaintext.

5.2 Domain Controller

The DC consists of four separate modules: the authentication service, the network service directory, and the topology and capability construction service in the Protection Layer Controller. For authentication purposes, the DC was preconfigured with the public keys of all switches.

Capability construction. For end-to-end path calculations when constructing capabilities, we use a bidirectional search from both the source and destination. All computed routes are cached at the DC to speed up subsequent capability requests for the same pair of end hosts, although cached routes are checked against the current topology to ensure freshness before re-use.

Capabilities use 8-bit IDs to denote the incoming and outgoing switch ports. Switch IDs are 32 bits and the service IDs are 16 bits. The innermost layer of the capability requires 24 bytes, while each additional layer uses 14 bytes. The longest path on our test topologies was 10 switches in length, resulting in a 164 byte header.

Service Directory. DNS queries for all unauthenticated users on our network resolve to the DC's IP address, which hosts a simple webserver. We provide a basic HTTP interface to the service directory. Through a web browser, users can log in via a simple web-form and can then browse the service directory or, with the appropriate permissions, perform other operations (such as adding and deleting services).

The directory service also provides an interface for managing users and groups. Non-administrative users are able to create their own groups and use them in access-control declarations.

To access a service, a client browses the directory tree for the desired service, each of which is listed as a link. If a service is selected, the directory server checks the user's permissions. If successful, the DC generates capabilities for the flows and sends them to the client (or more accurately, the client's SANE IP proxy). The web-server returns an HTTP redirect to the service's appropriate protocol and network address, e.g., `ssh://192.168.1.1:22/`. The client's browser can then launch the appropriate application if one such is registered; otherwise, the user must do so by hand.

5.3 Example Operation

As a concrete example, we describe the events for an ssh session initiated within our internal network. All participating end hosts have a translation proxy positioned between them and the rest of the network. Additionally, they are configured so that the DC acts as their default DNS server.

Until a user has logged in, the translation proxy returns the DC's IP address for all DNS queries and forwards all TCP packets sent to port 80 to the DC. Users opening a web-browser are therefore automatically forwarded to the DC so that they may log in. This is similar in feel to admission control systems employed by hotels and wireless access points. All packets forwarded to the DC are accompanied by a SANE header which is added by the translation proxy. Once a user has authenticated, the DC caches the user's location (derived from the SANE header of the authentication packets) and associates all subsequent packets from that location with the user.

Suppose a user ssh's from machine A to machine B. A will issue a DNS request for B. The translation proxy will intercept the DNS packet (after forging an ARP reply) and translate the DNS requests to a capability request for machine B. Because the the DNS name does not contain an indication of the service, by convention we prepend the service name to the beginning of the DNS request (e.g. ssh ssh.B.stanford.edu). The DC does the permission check based on the capability (initially added by the translation proxy) and the ACL of the requested service.

If the permission check is successful, the DC returns the capabilities for the client and server, which are cached at the translation proxy. The translation proxy then sends a DNS reply to A with a unique destination IP address d, which allows it to demultiplex subsequent packets. Subsequently, when the translation proxy receives packets from A destined to d, it changes d to the destination's true IP address (much like a NAT) and tags the packet with the appropriate SANE capability. Additionally, the translation proxy piggybacks the return capability destined for the server's translation proxy on the first packet. Return traffic from the server to the client is handled similarly.

6 Evaluation

We now analyze the practical implications of running SANE on a real network. First, we study the performance of our software implementation of the DC and switches. Next, we use packets traces collected from a medium-sized network to address scalability concerns and to evaluate the need for DC replication.

6.1 Microbenchmarks

Table 1 shows the performance of the DC (in capabilities per second) and switches (in Mb/s) for different capability packet sizes (i.e., varying average path lengths). All tests were done on a commodity 2.3GHz PC.

As we show in the next section, our naive implementation of the DC performs orders of magnitude better than is necessary to handle request traffic in a medium-sized enterprise.

The software switches are able to saturate the 100Mb/s network on which we tested them. For larger capability sizes, however, they were computationally-bound by decryption—99% of CPU time was spent on decryption alone—leading to poor throughput performance. This is largely due to the use of an unoptimized encryption library. In practice, SANE switches would be implemented in hardware. We note that modern switches, such as Cisco's catalyst 6K family, can perform MAC level encryption at 10Gb/s. We are in the process of reimplementing SANE switches in hardware.

6.2 Scalability

One potential concern with SANE's design is the centralization of function at the Domain Controller. As we discuss in Section 3.5, the DC can easily be physically replicated. Here, we seek to understand the extent to which replication would be necessary for a medium-sized enterprise environment, basing on conclusions on traffic traces collected at the Lawrence Berkeley National Laboratory (LBL) [36].

The traces were collected over a 34-hour period in January 2005, and cover about 8,000 internal addresses. The trace's anonymization techniques [37] ensure that (1) there is an isomorphic mapping between hosts' real IP addresses and the published anonymized addresses, and (2) real port numbers are preserved, allowing us to identify the application-level protocols of many packets. The trace contains almost 47 million packets, which includes 20,849 DNS requests and 145,577 TCP connections.

Figure 6 demonstrates the DNS request rate, TCP connection establishment rate, and the maximum number of concurrent TCP connections per second, respectively.

The DNS and TCP request rates provide an estimate for an expected rate of DC requests by end hosts in a SANE network. The DNS rate provides a lower-bound that takes client-side caching into effect, akin to SANE end hosts multiplexing multiple flows using a single capability, while the TCP rate provides an upper bound.

	5 hops	10 hops	15 hops
DC	100,000 cap/s	40,000 cap/s	20,000 cap/s
switch	762 Mb/s	480 Mb/s	250 Mb/s

Table 1: Performance of a DC and switches

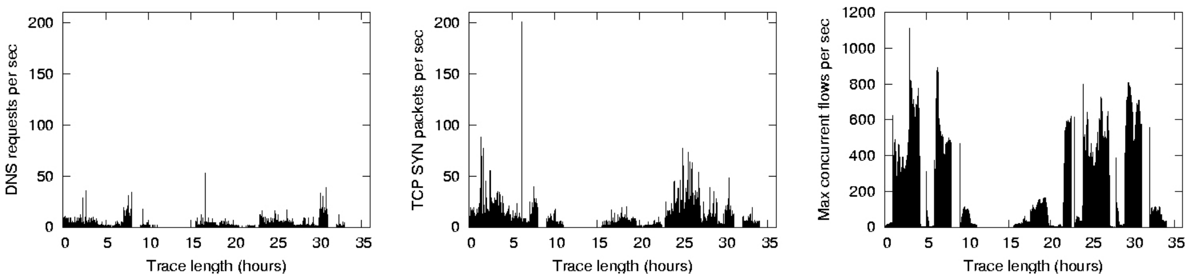


Figure 6: DNS requests, TCP connection establishment requests, and maximum concurrent TCP connections per second, respectively, for the LBL enterprise network.

Even for this upper bound, we found that the peak rate was fewer than 200 requests per second, which is 200 times lower than what our unoptimized DC implementation can handle (see Table 1).

Next, we look at what might happen upon a link failure, whereby all end hosts communicating over the failed link simultaneously contact the DC to establish a new capability. To understand this, we calculated the maximum concurrent number of TCP connections in the LBL network.[9] We find that the dataset has a maximum of 1,111 concurrent connections, while the median is only 27 connections. Assuming the worst-case link failure—whereby all connections traverse the same network link which fails—our simple DC can still manage 40 times more requests.

Based on the above measurements, we estimate the bandwidth consumption of control traffic on a SANE network. In the worst case, assuming no link failure, 200 requests per second are sent to the DC. We assume all flows are long-lived, and that refreshes are sent every 10 minutes. With 1,111 concurrent connections in the worst case, capability refresh requests result in at most an additional 2 packets/s.[10] Given header sizes in our prototype implementation and assuming the longest path on the network to be 10 hops, packets carrying the forward and return capabilities will be at most 0.4 KB in size, resulting in a maximum of 0.646 Mb/s of control traffic.

This analysis of an enterprise network demonstrates that only a few domain controllers are necessary to handle DC requests from tens of thousands of end hosts. In fact, DC replication is probably more relevant to ensure uninterrupted service in the face of potential DC failures.

7 Related Work

Network Protection Mechanisms. Firewalls have been the cornerstone of enterprise security for many years. However, their use is largely restricted to enforcing coarse-grain network perimeters [45]. Even in this limited role, misconfiguration has been a persistent problem [46, 47]. This can be attributed to several factors which SANE tries to address; in particular, their low-level policy specification and very localized view leaves firewalls highly sensitive to changes in topology. A variety of efforts have examined less error prone methods for policy specification [13], as well as how to detect policy errors automatically [33].

The desire for a mechanism that supports ubiquitous enforcement, topology independence, centralized management, and meaningful end-point identifiers has lead to the development of distributed firewalls [14, 26, 2]. Distributed firewalls share much with SANE in their initial motivation but differ substantially in their trust and usage model. First, they require that some software be installed on the end host. This can be beneficial as it provides greater visibility into end host behavior, however, it comes at the cost of convenience. More importantly, for end hosts to perform enforcement, that end host must be trusted (or at least some part of it, e.g., the OS [26], a VMM [22], the NIC [31], or some small peripheral [40]). Furthermore, in a distributed firewall scenario, the network infrastructure itself receives no protection, i.e., the network is still "on" by default. This design affords no defense-in-depth if the end-point firewall is bypassed, as it leaves all other network elements (e.g., switches, middleboxes, and unprotected end hosts) exposed.

Weaver et al. [45] argue that existing configurations of coarse-grain network perimeters (e.g., NIDS and multiple firewalls) and end host protective mechanisms (e.g. anti-virus software) are ineffective against worms, both when employed individually or in combination. They advocate augmenting traditional coarse-grain perimeters with fine-grain protection mechanisms throughout the network, especially to detect and halt worm propagation.

Finally, commercial offerings from Consentry [3] introduce special-purpose bridges for enforcing access control policy. To our knowledge, these solutions require that the bridges be placed at a choke point in the network so that all traffic needing enforcement passes through them. In contrast, SANE permission checking is done at a central point only on connection setup, decoupling it from the data path. SANE's design both allows redundancy in the network without undermining network security policy and simplifies the forwarding elements.

Dealing with Routing Complexity. Often misconfigured routers make firewalls simply irrelevant by routing around them. The inability to reason about connectivity in complex enterprise networks has fueled commercial offerings such as those of Lumeta [5], to help administrators discover what connectivity exists in their network.

In their 4D architecture, Rexford et al. [41, 24] argue that the decentralized routing policy, access control, and management has resulted in complex routers and cumbersome, difficult-to-manage networks. Similar to SANE, they argue that routing (the control plane) should be separated from forwarding, resulting a very simple data path. Although 4D centralizes routing policy decisions, they retain the security model of today's networks. Routing (forwarding tables) and access controls (filtering rules) are still decoupled, disseminated to forwarding elements, and operate the basis of weakly-bound end-point identifiers (IP addresses). In our work, there is no need to disseminate forwarding tables or filters, as forwarding decisions are made *a priori* and encoded in source routes.

Predicate routing [43] attempts to unify security and routing by defining connectivity as a set of declarative statements from which routing tables and filters are generated. SANE differs, however, in that users are first-class objects—as opposed to end-point IDs or IP addresses in Predicate routing—and thus can be used in defining access controls.

Expanding the Link-layer. Reducing a network from two layers of connectivity to one, where all forwarding is done entirely at the link layer, has become a popular method of simplifying medium-sized enterprise networks. However, large Ethernet-only networks face significant problems with scalability, stability, and fault tolerance, mainly due to their use of broadcast and spanning-tree-based forwarding.

To address these concerns, several proposals have suggested replacing the MST-based forwarding at the link-layer with normal link-state routing [39, 35]. Some, such as Myers et al. [35], advocate changing the Ethernet model to provide explicit host registration and discovery based on a directory service, instead of the traditional broadcast discovery service (ARP) and implicit MAC address learning. This provides better scalability and transparent link-layer mobility, and it eliminates the inefficiencies of broadcast. Similarly, SANE eliminates broadcast in favor of tighter traffic control through link-state updates and source routes. However, we eschew the use of persistent end host identifiers, instead associating each routable destination with the switch port from where it registered.

Capabilities for DDOS prevention. Much recent work has focused on DoS remediation through network enforced capabilities on the WAN [12, 51, 52]. These systems assumes no cooperation between network elements, nor do they have a notion of centralized control. Instead, clients receive capabilities from servers directly and vice versa. Capabilities are constructed on-route by the initial capability requests. This offers a very different policy model than SANE, as it is designed to meet different needs (limiting wide area DoS) and relies on different operating assumptions (no common administrative domain).

8 Conclusion

We believe that enterprise networks are different from the Internet at large and deserve special attention: Security is paramount, centralized control is the norm, and uniform, consistent policies are important. However, providing strong protection is difficult, and it requires some tradeoffs. There are clear advantages to having an open environment where connectivity is unconstrained and every end host can talk to every other. Just as clearly, however, such openness is prone to attack by malicious users from inside or outside the network.

We set out to design a network that greatly limits the ability of an end host or switch to launch an effective attack, while still maintaining flexibility and ease of management. Drastic goals call for drastic measures, and we understand that our proposal—SANE—is an extreme approach. SANE is conservative in the sense that it gives the least possible privilege and knowledge to all parties, except to a trusted, central Domain Controller. We believe that this would be an acceptable practice in enterprises, where central control and restricted access are common.

Yet SANE remains practical: Our implementation shows that SANE could be deployed in current networks with only a few modifications, and it can easily scale to networks of tens of thousands of nodes.

9 Acknowledgements

We would like to thank Mendel Rosenblum, Vern Paxson, Nicholas Weaver, Mark Allman and Bill Cheswick for their helpful comments on this project. We also like to thank the anonymous reviewers for their feedback and especially our shepherd, Michael Roe, for his guidance. This research was supported in part by the Stanford Clean Slate program, the 100x100 project and NSF. Part of this research was performed while on appointment as a U.S. Department of Homeand Security (DHS) Fellow under the DHS Scholarship and Fellowship Program, a program administered by the Oak Ridge Institute for Science and Education (ORISE) for DHS through an interagency agreement with the U.S Department of Energy (DOE). ORISE is managed by Oak Ridge Associated Universities under DOE contract number DE-AC05-00OR22750. All opinions expressed in this paper are the authors' and do not necessarily reflect the policies and views of DHS, DOE, ORISE, or NSF. This work was also supported in part by TRUST (The Team for Research in Ubiquitous Secure Technology), which receives support from the National Science Foundation (NSF award number CCF-0424422).

Notes

[1]A policy might be specified by many people (e.g, LDAP), but is typically centrally managed.

[2]SANE is agnostic to the PKI or other authentication mechanism in use (e.g. Kerberos, IBE). Here, we will assume principals and switches have keys that have been certified by the enterprises CA.

[3]To establish shared keys, we opt for a simple key-exchange protocol from the IKE2 [28] suite.

[4]Request capabilities are similar to network capabilities as discussed in [12, 51]

[5]We use the same IV for all layers—as opposed to picking a new random IV for each layer—to reduce the capability's overall size. For standard modes of operation (such as CBC and counter-mode) reusing the IV in this manner does not impact security, since each layer uses a different symmetric key.

[6]For example, while SANE's protection layer prevents an adversary from targeting arbitrary switches, an attacker can attempt to target a switch indirectly by accessing an upstream server for which it otherwise has access permission.

[7]Normally, DC packet headers contain a consistent sender-ID in cleartext, much like the IPSec ESP header. This sender-ID tells the DC which key to use to authenticate and decrypt the payload. We replace this static ID with an ephemeral nonce provided by the DC. Every DC response contains a new nonce to use as the sender-ID in the next message.

[8]Implementing threshold cryptography for symmetric encryption is done combinatorially [16]: Start from a t-out-of-t sharing (namely, encrypt a DC master secret under all independent DC server keys) and then construct a t-out-of-n sharing from it.

[9]To calculate the concurrent number of TCP connections, we tracked srcip:srcport:dstip:dstport tuples, where a connection is considered finished upon receiving the first FIN packet or if no traffic packets belonging to that tuple are seen for 15 minutes. There were only 143 cases of TCP packets that were sent after a connection was considered timed-out.

[10]This is a conservative upper bound: In our traces, the average flow length is 92s, implying that at most, 15% of the flows could have lengths greater than 10 minutes.

References

[1] 802.1D MAC Bridges. http://www.ieee802.org/1/pages/802.1D-2003.html.

[2] Apani home page. http://www.apani.com/.

[3] Consentry home page. http://www.consentry.com/.

[4] DNS Service Discover (DNS-SD). http://www.dns-sd.org/.

[5] Lumeta. http://www.lumeta.com/.

[6] UPnP Standards. http://www.upnp.org/.

[7] Cisco Security Advisory: Cisco IOS Remote Router Crash. http://www.cisco.com/warp/public/770/ioslogin-pub.shtml, August 1998.

[8] CERT Advisory CA-2003-13 Multiple Vulnerabilities in Snort Preprocessors. http://www.cert.org/advisories/CA-2003-13.html, April 2003.

[9] Sasser Worms Continue to Threaten Corporate Productivity. http://www.esecurityplanet.com/alerts/article.php/3349321, May 2004.

[10] Technical Cyber Security Alert TA04-036Aarchive HTTP Parsing Vulnerabilities in Check Point Firewall-1. http://www.us-cert.gov/cas/techalerts/TA04-036A.html, February 2004.

[11] ICMP Attacks Against TCP Vulnerability Exploit. http://www.securiteam.com/exploits/5SP0N0AFFU.html, April 2005.

[12] T. Anderson, T. Roscoe, and D. Wetherall. Preventing Internet Denial-of-Service with Capabilities. *SIGCOMM Comput. Commun. Rev.*, 34(1):39–44, 2004.

[13] Y. Bartal, A. J. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. *ACM Trans. Comput. Syst.*, 22(4):381–420, 2004.

[14] S. M. Bellovin. Distributed firewalls. *;login:*, 24(Security), November 1999.

[15] M. Blaze, J. Feigenbaum, and A. D. Keromytis. Keynote: Trust management for public-key infrastructures (position paper). In *Proceedings of the 6th International Workshop on Security Protocols*, pages 59–63, London, UK, 1999. Springer-Verlag.

[16] E. Brickell, G. D. Crescenzo, and Y. Frankel. Sharing block ciphers. In *Proceedings of Information Security and Privacy*, volume 1841 of *LNCS*, pages 457–470. Springer-Verlag, 2000.

[17] M. Casado and N. McKeown. The Virtual Network System. In *Proceedings of the ACM SIGCSE Conference*, 2005.

[18] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, Nov. 2002.

[19] D. Cullen. Half Life 2 leak means no launch for Christmas. http://www.theregister.co.uk/2003/10/07/half_life_2_leak_means/, October 2003.

[20] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In *Advances in Cryptology - Crypto '89*, 1990.

[21] J. R. Douceur. The Sybil attack. In *First Intl. Workshop on Peer-to-Peer Systems (IPTPS 02)*, Mar. 2002.

[22] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)*, October 2003.

[23] D. M. Goldschlag, M. G. Reed, and P. F. Syverson. Hiding Routing Information. In R. Anderson, editor, *Proceedings of Information Hiding: First International Workshop*, pages 137–150. Springer-Verlag, LNCS 1174, May 1996.

[24] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A Clean Slate 4D Approach to Network Control and Management. In *In ACM SIGCOMM Computer Communication Review*, October 2005.

[25] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, Feb. 1988.

[26] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a distributed firewall. In *ACM Conference on Computer and Communications Security*, pages 190–199, 2000.

[27] G. C. S. Jian Pu, Eric Manning. Routing Reliability Analysis of Partially Disjoint Paths. In *IEEE Pacific Rim Conference on Communications, Computers and Signal processing (PACRIM' 01)*, volume 1, pages 79–82, August 2001.

[28] C. Kaufman. Internet key exchange (ikev2) protocol. draft-ietf-ipsec-ikev2-10.txt (Work in Progress).

[29] A. Kumar, V. Paxson, and N. Weaver. Exploiting underlying structure for detailed reconstruction of an internet-scale event. In *to appear in Proc. ACM IMC*, October 2005.

[30] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. Socks protocol version 5. RFC 1928, March 1996.

[31] T. Markham and C. Payne. Security at the Network Edge: A Distributed Firewall Architecture. In *DARPA Information Survivability Conference and Exposition*, 2001.

[32] G. M. Marro. Attacks at the data link layer, 2003.

[33] A. Mayer, A. Wool, and E. Ziskind. Fang: A firewall analysis engine. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 177, Washington, DC, USA, 2000. IEEE Computer Society.

[34] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. *IEEE Security and Privacy*, 1(4):33–39, 2003.

[35] A. Myers, E. Ng, and H. Zhang. Rethinking the Service Model: Scaling Ethernet to a Million Nodes. In *ACM SIGCOMM HotNets*, November 2004.

[36] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney. A first look at modern enterprise traffic. In *ACM/USENIX Internet Measurement Conference*, Oct. 2005.

[37] R. Pang, M. Allman, V. Paxson, and J. Lee. The devil and packet trace anonymization. *ACM Comput. Commun. Rev.*, 36(1), Jan. 2006.

[38] R. Perlman. *Network layer protocols with Byzantine robustness*. PhD thesis, Massachussets Institute of Technology, 1988.

[39] R. J. Perlman. Rbridges: Transparent Routing. In *INFOCOM*, 2004.

[40] V. Prevelakis and A. D. Keromytis. Designing an Embedded Firewall/VPN Gatweway. In *Proc. International Network Conference*, July 2002.

[41] J. Rexford, A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, G. Xie, J. Zhan, and H. Zhang. Network-Wide Decision Making: Toward A Wafer-Thin Control Plane. In *Proceedings of HotNets III*, November 2004.

[42] P. Rogaway, M. Bellare, J. Black, and T. Krovetz. OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption. In *ACM Conference on Computer and Communications Security*, pages 196–205, 2001.

[43] T. Roscoe, S. Hand, R. Isaacs, R. Mortier, and P. Jardetzky. Predicate routing: Enabling controlled networking. *SIGCOMM Comput. Commun. Rev.*, 33(1):65–70, 2003.

[44] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. Service location protocol. RFC 2165, july 1997.

[45] N. Weaver, D. Ellis, S. Staniford, and V. Paxson. Worms vs. Perimeters: The Case for Hard-LANs. In *Proc. Hot Interconnects 12*, August 2004.

[46] A. Wool. A quantitative study of firewall configuration errors. *IEEE Computer*, 37(6):62–67, 2004.

[47] A. Wool. The use and usability of direction-based filtering in firewalls. *Computers & Security*, 26(6):459–468, 2004.

[48] S. Wu, B. Vetter, and F. Wang. An experimental study of insider attacks for the OSPF routing protocol. October 1997.

[49] G. Xie, J. Zhan, D. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of ip networks. In *IEEE INFOCOM 2005*, March 2005.

[50] G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, and G. Hjalmtysson. Routing design in operational networks: A look from the inside. In *Proc. ACM SIGCOMM '04*, pages 27–40, New York, NY, USA, 2004. ACM Press.

[51] A. Yaar, A. Perrig, and D. Song. Siff: A stateless internet flow filter to mitigate ddos flooding attacks. In *In Proceedings of the IEEE Security and Privacy Symposium*, May 2004.

[52] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. In *Proc. ACM SIGCOMM '05*, pages 241–252, New York, NY, USA, 2005. ACM Press.

PHAS: A Prefix Hijack Alert System [*]

Mohit Lad [†]
mohit@cs.ucla.edu

Dan Massey [‡]
massey@cs.colostate.edu

Dan Pei [§]
peidan@research.att.com

Yiguo Wu [†]
yiguowu@cs.ucla.edu

Beichuan Zhang [¶]
bzhang@cs.arizona.edu

Lixia Zhang [†]
lixia@cs.ucla.edu

Abstract

In a BGP prefix hijacking event, a router originates a route to a prefix, but does not provide data delivery to the actual prefix. Prefix hijacking events have been widely reported and are a serious problem in the Internet. This paper presents a new Prefix Hijack Alert System (PHAS). PHAS is a real-time notification system that alerts prefix owners when their BGP origin changes. By providing reliable and timely notification of origin AS changes, PHAS allows prefix owners to quickly and easily detect prefix hijacking events and take prompt action to address the problem. We illustrate the effectiveness of PHAS and evaluate its overhead using BGP logs collected from RouteViews. PHAS is light-weight, easy to implement, and readily deployable. In addition to protecting against false BGP origins, the PHAS concept can be extended to detect prefix hijacking events that involve announcing more specific prefixes or modifying the last hop in the path.

1 Introduction

The Internet relies on the Border Gateway Protocol (BGP)[16] to convey routing information. However, if BGP provides incorrect routing information, packets may never reach the intended destination, and may even be misdirected to malicious destinations. The inability to ensure the integrity and correctness of routing information leads to many known vulnerabilities in BGP [12]. This paper considers a simple and common vulnerability called prefix hijacking. In a common prefix hijacking event, an Autonomous System (AS) originates a route for an address space, termed as prefix, but does not provide data delivery for that prefix. In other words, an AS reports "use me to reach prefix p", but does not actually provide data delivery for prefix p. For example, on December 24, 2004, AS 9121 incorrectly originated routes to 106,089 prefixes, almost 70% of all the prefixes. BGP routers throughout the Internet selected the route originating from AS 9121 as the best path to some or all of these prefixes. Traffic for these prefixes was then forwarded to AS 9121, who then essentially dropped the packets, affecting thousands of organizations [13]. When a prefix is hijacked, sensitive data from unsuspecting users could easily fall into the wrong hands, resulting in serious security and privacy breaches. A recent study has also found that spammers hijack BGP prefixes to send spam mail [14]. Thus, prefix hijacking is real operational concern in the Internet, and securing Internet routing against prefix hijacking is an important problem.

Simply detecting the occurrence of a prefix hijack event is an essential, but difficult task. Large-scale events where an AS mistakenly hijacks thousands of prefixes may be detected relatively quickly due to their size and impact. For example, in the AS 9121 event described above, thousands of prefixes from different origins, suddenly changed to origin AS 9121, a clear indication of prefix hijack. But smaller scale errors and intentional attacks can be much more difficult to detect. For example, suppose a malicious AS originates a false path to only one prefix, 131.179.0.0/16 (UCLA). Some BGP routers will accept the new false path while others may continue to use the correct path originated by UCLA. An origin change for a single prefix is a common occurrence and is unlikely to trigger alarm. As we will show later in the paper, there are quite a few origin AS changes during a typical day and most of these changes are valid. A prefix may change its origin AS at any time due to contrac-

[*]This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No N66001-04-1-8926 and by National Science Fundation(NSF) under Contract No ANI-0221453. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the DARPA or NSF.

[†]Computer Science Department, UCLA

[‡]Computer Science Department, Colorado State University

[§]AT&T Labs–Research

[¶]Computer Science Department, University of Arizona tual arrangement, multi-homing, traffic engineering, and a host of other factors. Only the origin itself (UCLA in our example) could easily and accurately distinguish between a legitimate origin change and a prefix hijack [4]. The legitimate origin is best able to identify this type of prefix hijack, but it has very little information about the BGP routes taken by others to its own prefix. In this case, UCLA may notice a drop in traffic and/or reports of connectivity problems, but there are numerous potential causes for this. Even if UCLA suspected a prefix hijacking attack, UCLA's local data can only confirm that it has correctly announced its own route. To determine if others are incorrectly announcing a route to UCLA, the UCLA administrators would need data from other remote sites.

One of the goals behind the existing BGP monitoring projects such as Oregon RouteViews and RIPE RRC is to provide network operators with a remote view of their own prefixes. Through establishing BGP peering with operational routers, the RouteViews and RIPE RRC projects collect routing data from a few hundred BGP routers around the globe placed in critical exchange points, tier-1 ISPs, and so forth.[1] These BGP data collectors obtain information in real time, which can be used to quickly detect prefix hijacking and identify the source of the problem. For example, a prefix hijack event occurred on January 22, 2006 and affected close to 80 prefixes including a financial organization. Within a few seconds of the event occurrence, RouteViews data collector received update messages from several of its BGP peers indicating a new origin to the prefix of the financial organization. If the prefix owner had received this data, it could have immediately detected the prefix hijacking and could have quickly taken corrective measures using operator channels. However, prefix owners do not have any way to easily access the data. The current BGP monitors collect vast amounts of data and dump the raw data, unsorted, onto the disk. It is impractical to assume that all the prefix owners would be able to download this the data and then extract the information about their own prefixes, let alone doing so in real-time.

In this paper, we build on the premise that the prefix owner is the only one who can accurately distinguish between legitimate changes and prefix hijacking events, and propose a scalable system for providing prefix owners with timely and reliable notifications of potential prefix hijacks. During a prefix hijack, the notification itself may reach the hijacker instead of the prefix owner, and thus the prefix owner would not be informed of the ongoing hijack. To increase the chances of notification delivery, we use a multi-path delivery mechanism using the existing email infrastructure to increase the chances of notification delivery. Our design is readily deployable and easy to use. Once our system has detected the problem, the owner can then take necessary actions, including soliciting help through operator channels like North American Network Operators Group (NANOG) mailing lists, and the NSP-Security mailing lists to either resolve the problem with the hijacker or its upstream ISPs.

The remainder of the paper is organized as follows. Section 2 presents some basics about routing and prefix hijacks. Section 3 presents our system design in detail. Notification generation, notification delivery, and local notification filtering are presented in Section 4, Section 5, and Section 6, respectively. Section 7 evaluates our design. Section 8 shows how the detection capability of our system can be extended to handle other forms of prefix hijacking. Section 9 reviews related work and Section 10 concludes the paper.

2 Background

The Internet consists of a large number of networks called Autonomous Systems (AS), and BGP (Border Gateway Protocol [15]) is the protocol used to exchange routing information between these ASes. Each AS is represented by a unique numeric ID and destinations are in the form of prefixes, where each prefix represents a network space. For example, a prefix 131.179.96.0/24 represents a network space of 2^8 addresses belonging to AS 52 (UCLA). Authorities such as ARIN and RIPE assign prefixes to organizations, who then become the owner of the prefixes.

As a path vector routing protocol, BGP lists the entire AS path to reach a destination prefix in its routing updates. In Figure 1(a), AS 52 (UCLA) is the owner of prefix 131.179.0.0/16 and announces this prefix to its upstream service providers. The AS announcing a prefix to the rest of the Internet, is called the origin AS of the prefix. In this example, AS 52 is the origin AS of prefix 131.179.0.0/16.[2] An AS receiving a path to reach a prefix may choose to propagate the path to some or all of its neighbors. An AS intending to propagate a received path, prepends its own AS ID to the path before sending the announcement to its neighbors. Therefore, in Figure 1(a), AS A has an AS path of (A, P, Q, R, 52) for prefix 131.179.0.0/16 and AS B has an AS path of (B, R, 52). When multiple prefixes cover the same address, the longest prefix match rule is used to forward the traffic. Thus, if a BGP routing table contains paths to reach prefix 131.179.0.0/24 as well as 131.179.0.0/16, then a

[1]Admittedly a few hundred routers represent only a small fraction of the overall Internet. A prefix hijack that affects only a small local region may not be observed by any of the current BGP monitors. In a separate project, we are studying the optimal BGP monitor placement problem, however those results are beyond the scope of this paper.

[2]Sometimes the owner of a prefix might not run BGP and its provider AS serves as the prefix origin.

packet destined to 131.179.0.128 would choose the path to reach 131.179.0.0/24.

2.1 Prefix Hijacking

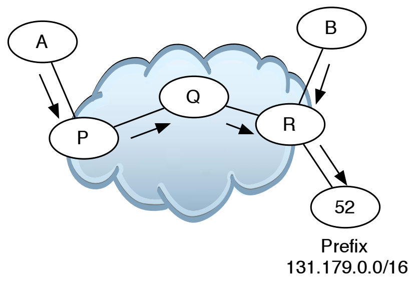
In a prefix hijack event, the announced path to the prefix cannot actually be used to deliver data to the prefix. In some parts of the Internet, the false path replaces the authentic route to the prefix and traffic that follows the false path will eventually be dropped or delivered to someone who is pretending to be the legitimate destination. In other words, the traffic sent along the false path has been hijacked. We term the AS injecting false information as an attacker AS, and the AS that owns the route as a victim AS. For example, in Figure 1b, AS 110 announces 131.179.0.0/16, while the true origin for this prefix should be AS 52. It can be seen in this example, that AS 110 successfully effected a hijack, since AS A decided to pick the route to AS 110 instead of AS 52. In this case, AS 52 is the victim, AS 110 is the attacker and any traffic sent by AS A is delivered to AS 110 rather than the legitimate origin. Note that AS 52 may see a drop in its overall traffic volume, but variations in traffic load are the norm for most networks and AS 52 may be completely unaware that hijack event is occurring.

An attacker AS can hijack a prefix in various ways such as falsely announcing itself as the origin for a prefix (as discussed in the example above), falsely modifying some portion of the path other than origin, or falsely announcing a more specific prefix. Our presentation of PHAS is particularly concerned with the first case case where the origin AS is not valid. Section 8 discusses how the PHAS concept can be extended to handle path modifications adjacent to the origin AS and announcement of more specific prefixes.

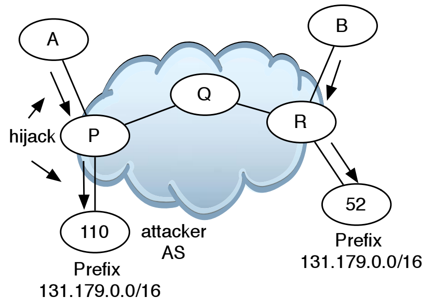
3 System Design

Our basic approach is to examine BGP routing data collected at RouteViews (or RIPE, or any other BGP collectors), and provide real time notifications of any potential prefix hijacking to the prefix owner in a reliable way. In particular, we should immediately notify the prefix owner anytime a new origin AS is associated with their prefix. At a potentially slower rate, the prefix owner should be notified when an origin AS is no longer used to announce its prefix. The net result is that the prefix owner is able to track the set of AS numbers that originate its prefix. Presumably, the prefix owner also knows which AS numbers are allowed to serve as origins and can thus detect any false origins, as well as know when the false origins have stopped announcing its prefix.

More formally, we define an *origin set* for each prefix and track changes of this origin set. Existing BGP



a. True origin AS 52 announces prefix 131.179.0.0/16



b. False origin AS 110 announces prefix 131.179.0.0/16 and hijacks A's route

Figure 1: Example of prefix hijack

monitoring projects such as RouteViews and RIPE, peer with a few hundred BGP routers around the globe and collect BGP updates in real-time. Each monitored BGP router, or monitor in short, reports its best path to a prefix P and the last hop in this path is an origin AS for P. We define the origin set $O_{SET}(P, t)$ for a prefix P as the union of all the origins seen by all the monitors for that prefix at time t. For example, on January 22, 2006 before 8:31 hours GMT, all RouteViews monitors reported paths ending in AS 19758 for prefix 65.173.134.0/24, and thus for time $t <$ 8:31 on 01/22/2006, $O_{SET}(65.173.134.0/24, t) = \{19758\}$. When the prefix was hijacked at 8:31AM, some monitors switched to paths ending with AS 27506 and thus for the time t = 8:31 on 1/22/2006, $O_{SET}(65.173.134.0/24, t) = \{19758, 27506\}$. Our objective is to immediately notify the owner of 65.173.134.0/24 of this new origin set, and the owner could then work to resolve this issue with the offending AS 27506 or its upstream providers. Later, when the origin AS 27506 would not be seen as announcing this prefix anymore, we would like to send a notification to the prefix owner indicating that the origin set $O_{SET}(65.173.134.0/24, t) = \{19758\}$, so that the prefix owner also knows that the problem has been resolved.

Our design consists of the following four components.

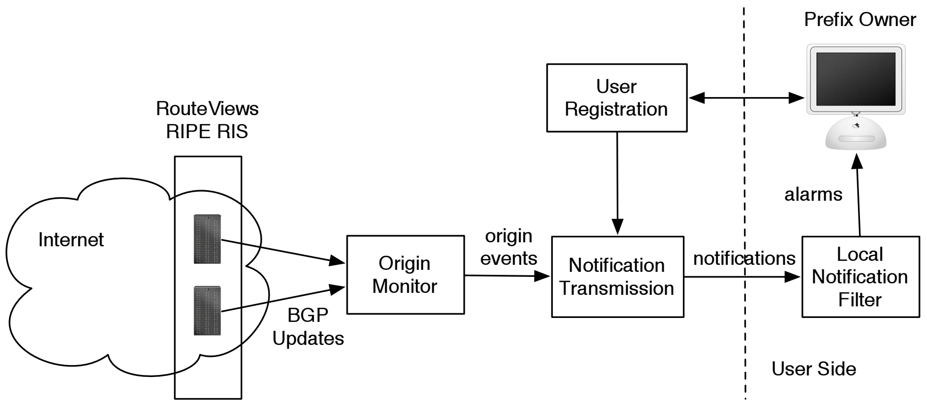


Figure 2: Components of PHAS

1. *User Registration:* All prefix owners who are interested in using our system need to register with the PHAS server and provide contact email addresses. PHAS aims to provide a web-based registration service, similar to the standard mailing list registration process. Each new user opens an account by his/her email address and a password via a secure HTTPS session. This action is sent to the email address for confirmation. Once confirmed, the registration is committed, and any later change to the account is done via HTTPS and requires the password. The registration specifies which prefixes are of interest and each registrant is strongly encouraged to submit multiple email addresses hosted by different email systems (such as a GMail address), to maximize the chance of email reception in face of prefix hijacking. Ideally, only the legitimate owner of a prefix should register, but verifying the correct contact address for each prefix is a challenging problem in its own right with no immediately deployable solution. In PHAS, an attacker may register and falsely claim to be the prefix owner. However, this action does not cancel the registration by the legitimate owner and all notifications are based on publicly available data so the attacker gains no new information by successfully registering.

2. *Origin Set Monitoring:* Using the BGP monitor data, PHAS maintains a current origin set for each registered prefix. If there is a change to this origin set, an origin event is generated.To control the number of origin events for prefixes with frequent origin changes, we use a time-window based mechanism to reduce the *repeated reporting* of the origin changes but still guarantee the immediate notification for any new origin announced for a prefix. We increase the duration of this window for prefixes that report lot of origin changes even after the default time window is used. The window duration is decreased if the number of origin event reduces. This adaptive window scheme is central to ensuring the system scales from the perspective of the origin set monitoring and also limits the number of false positives sent to the prefix owners. It is discussed further in Section 4.

3. *Notification Transmission:* Once an origin event is generated, PHAS decides whether the origin event translates to a notification message to be sent. For this, it checks the user registration to see if there are email addresses registered for the prefix involved. However, the seemingly simple task of sending a notification message, could be difficult in face of prefix hijack. For example, when the route to UCLA has been hijacked, email from PHAS to noc@ucla.edu may follow the hijacked route and never reach the intended receiver. To protect against this case, we strongly recommended two practices for prefix owners in order to set up “multiple diverse paths” for email delivery. First, in registering with our system, prefix owners should provide multiple email accounts on different email servers that are topologically diverse. Second, prefix owners should have Internet access via multiple prefixes. ISPs often have multiple prefixes of their own. For one that only owns a single prefix, a backup plan like a dial-up Internet access account is recommended. With the combination of multiple email addresses and multiple prefixes for Internet access, prefix owners can achieve a high success rate of notification delivery even in face of prefix hijack. All notification messages are also signed by PHAS server, whose public key is well-known. More details on the notification scheme will be discussed in Section 5.

4. *Local Notification Filter:* Although the notifications could be sent directly to network administrators, our

design assumes an automated processing of the received notifications. Tasks such as verifying the message is properly signed, checking whether periodic notifications has been received, and so forth are better handled by an automated receiver. In addition, many prefixes have multiple legitimate origins and thus not every change in the origin set is necessarily an attack that should be reported to the local network administrators. To make the system more user friendly, we provide a local filter program for processing the notification email. The local filter manages the external email addresses, checks any change in origin against a locally configured set of valid origins, and only reports an alarm to administrator when an unexpected origin change occurs. Local administrator can easily customize the filter program or even provide their own filter program. By incorporating a local filter, all the legitimate origin changes are simply screened out by the filter and only notices requiring human intervention are reported to the network operator. Local notification filter is discussed in more detail in Section 6.

Figure 2 shows the four components in our design and the interaction between them. Note how the origin events translate to notifications and finally to alarms.

4 Origin Change Detection

PHAS detects changes in BGP prefix origins and sends notification messages to registered prefix owners. For traffic engineering purposes, some networks may change their prefix origins frequently, which may trigger a large volume of notification messages if we want to keep track of every change. The main challenge in the system design is how to notify the owner in a timely manner while not being overwhelmed by the volume of messages.

4.1 Instantaneous Origin Changes

We first consider a simple scheme (Algorithm 1) that maintains an *origin set* for each prefix and sends a notification whenever the origin set changes. It takes input from a BGP monitoring project such as Route Views or RIPE. Let $\{M_1, M_2, ..., M_i, ..., M_N\}$ denote the set of N BGP routers providing data. By observing the BGP updates sent by router M_i, we can determine M_i's current route to prefix P. If M_i has a route to P at time t, $origin(M_i, P, t)$ denotes the origin AS of P in this route. If M_i has no route to P at time t, $origin(M_i, P, t)$ is empty. The *origin set* for prefix P at time t is defined as $O_{SET}(P, t) = \cup_{i=1}^{N} origin(M_i, P, t)$. In other words, the instantaneous origin set is simply the union of the origins currently used by any of the monitors to reach this prefix. As updates from M_i arrive, $origin(M_i, P, t)$ may change and thus the origin set may change as well. Whenever the origin set changes, we say an *origin event* is triggered.

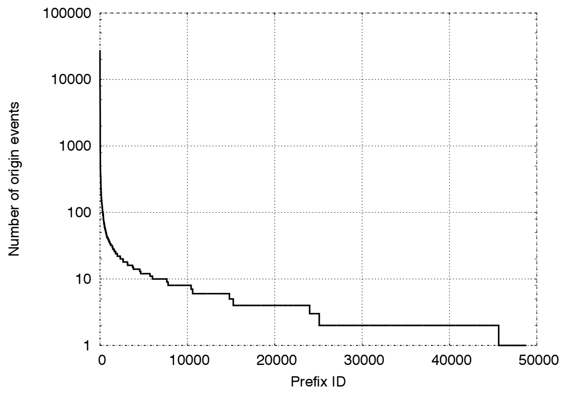


Figure 3: Origin events per prefix - December 2005

Algorithm 1: Instantaneous Origin Change

```
Initialize $origin(M_i, P, t_0)$ using the initial routing table of $M_i$ at time $t_0$;
$O_{SET}(P, t_0) = \cup_{i=1}^{N} origin(M_i, P, t_0)$;
if update for prefix $P$ at time $t$ from router $M_i$ is an announcement then
    $origin(M_i, P, t)$ = the last AS in the announced path;
else
    $origin(M_i, P, t) = \{\}$;
$O_{SET}(P, t) = \cup_{i=1}^{N} origin(M_i, P, t)$;
if $O_{SET}(P, t) \neq O_{SET}(P, t-1)$ then
    $origin\_gain = O_{SET}(P, t) - O_{SET}(P, t-1)$;
    $origin\_loss = O_{SET}(P, t-1) - O_{SET}(P, t)$;
    send $[O_{SET}(P, t), origin\_gain, origin\_loss]$ to prefix owner;
```

To study the algorithm behavior, we used data for the month of December 2005, from the RouteViews collector at the Oregon Internet Exchange. This BGP data collector peers with 42 operational routers from around the globe and thus the origin set is the union of the origin ASes seen by these 42 peers. The number of prefixes involved is close to 170,000. Algorithm 1 generated 511,513 origin events involving 48,768 prefixes during December 2005. Thus, close to 30% of the prefixes had one or more origin set changes. Figure 3 shows the distribution of the number of origin events per prefix (prefixes with no origin events are not plotted).

As the figure shows, some prefixes generated a large number of origin events. In Algorithm 1, even when the same origin leaves the set and comes back again

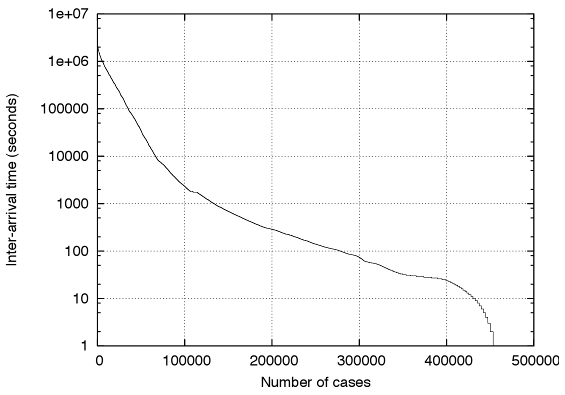


Figure 4: Inter-arrival time between origin events for a prefix for December 2005

on a repeated basis, each appearance and each disappearance triggers an origin event. For example, prefix 207.135.82.64/26 generated 5747 origin events during December 2005, simply due to the fact that its origin set switched frequently between $\{2828, 65000\}$, $\{2828\}$, and $\{65000\}$. Since some prefixes have unstable connectivity to the Internet, repeated withdrawal and announcement sequence causes the origin to frequently leave and join the set, resulting in repeated origin events. In order to detect prefix hijacking events, it is essential to immediately notify the owner when a new origin appears. However, reporting oscillations between already reported origins, as in this particular example, can be reduced.

4.2 Windowed Origin Changes

We now introduce the notion of windowed origin set. We can mask off repeated and frequent origin changes by reporting observed origin set over some time window, instead of reporting instantaneous origin set changes. Figure 4 plots the inter-arrival time between origin events. From the figure we can see that the inter-arrival time is less than 1000 seconds in close to 75% of the cases.

Let $O_{SET}(P, t, k)$ denote the set of all the origins for prefix P observed over the last k time units. In other words, this windowed origin set consists of all the origins for P that were observed by at least one router M_i during the time $[t-k, t]$. More formally, define $origin(M_i, P, t, k) = \cup_{i=t-k}^{t} origin(M_i, P, t)$ and $O_{SET}(P, k, t) = \cup_{i=1}^{N} origin(M_i, P, t, k)$. The definition includes the last k units at each time and thus provides a continuously moving window over which the origins of P are recorded. The algorithm to detect origin changes with a moving window is the same as Algorithm 1, except that we now have to include the time window k and only send origin events when $O_{SET}(P, t-1, k) \neq O_{SET}(P, t, k)$.

It is important to note that this revised algorithm only reduces the number of *repeated origin events*. The prefix owner will be immediately notified whenever a *new* (potentially false) origin appears for the first time during the last k time units. Suppose router M_i is the first to observe a new origin O for prefix P. If this new announcement first appears at time t, $Origin(M_i, P, t, k) = O$ and thus $O \in O_{SET}(P, t, k)$. Since M_i is the first to observe this origin, it must also be the case that $O \notin O_{SET}(P, t-1, k)$. Thus $O_{SET}(P, t-1, k) \neq O_{SET}(P, t, k)$ and an origin event is triggered at time t, i.e., as soon as the new origin appears. This feature guarantees timely detection and notification of potential prefix hijacking.

However, the addition of a time window does delay the notification of origin-loss events. Suppose origin O was in fact a prefix hijacking attempt. As discussed above, the prefix owner is immediately notified when O first appears. Assume as a result of this fast notification, the owner took actions and quickly resolved the attack. Let M_j denote the last monitored router to remove O from its routing table at time t_{end}. Although O has been removed from the routing tables, it will not be removed from M_j's origin set until time $t_{end}+k$. Thus O is also not removed from $Origin(M_i, P, t, k)$ until time $t_{end}+k$. The net result is that the prefix owner is not notified that O has been removed until k time units after O has vanished from the routing system.

4.3 Adaptive Window Size

Our objective is to reduce the number of repeated origin events for prefixes with frequent origin changes, but not penalize well-behaved prefixes by delaying reports that an origin has been removed. We start with a base time window of one hour. This masks transient changes for most prefixes, at a cost of delaying notification of origin loss events by one hour. However, some prefixes still generate a large number of notification messages even with the one hour window. Increasing the window size can further limit the number of repeated origin events for these prefixes but at the cost of further delaying origin-loss events for other prefixes. Rather than attempt to assign a uniform time window for all prefixes all the time, we introduce an adaptive window resizing scheme for each prefix. Essentially, prefixes that generate a large number of messages will be penalized by large window size, while other prefixes still use small window size.

Initially, each prefix starts with a penalty value of $penalty(P) = 0$ and a time window of one hour. Anytime a notification is generated for this prefix, $penalty(P)$ is increased by 0.5. The penalty value decays exponentially over time and the rate of decay is determined by a half-life parameter. We currently use a

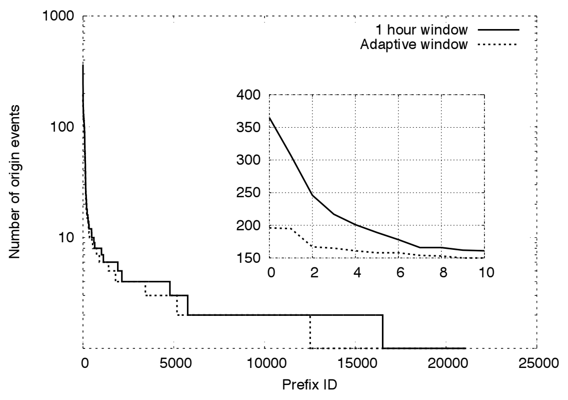


Figure 5: Distribution of origin events per prefix using adaptive window

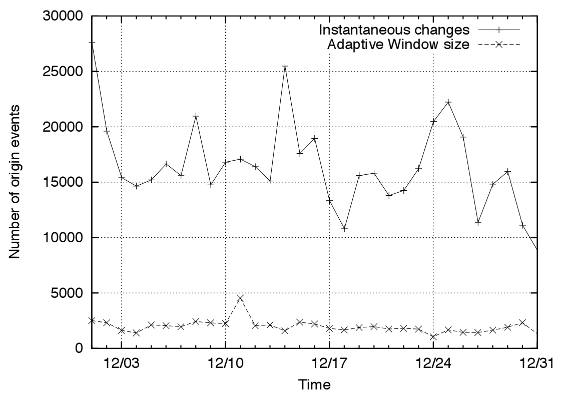


Figure 6: Comparison of origin events per day using instantaneous and adaptive window

half-life of 2 hours[3]. The size of the prefix's time window is set to $2^{\lfloor penalty(P) \rfloor}$ hours. In other words, a prefix with $penalty(P) < 1$ uses a time window of $2^0 = 1$ hour; a prefix with a $penalty(P)$ in the range $[1, 2)$ uses a time window of $2^1 = 2$ hours; a prefix with $penalty(P)$ in range $[2, 3)$ uses a time window of $2^2 = 4$ hours; and so forth.

Figure 5 shows the distribution of origin events generated using this adaptive window. For comparison, we also show the distribution using a fixed window size of 1 hour and show a zoomed in portion of the plot for the top 10 most active prefixes. Figure 6 shows the number of origin events generated per day using adaptive windows with a default as 1 hour along with the number of origin events using instantaneous origin changes for comparison. The introduction of the adaptive window reduces the number of origin events due to unstable prefixes, while still ensuring that any newly announced origin is immediately reported to the prefix owner. Prefixes that experience large number of origin changes would experience a longer delay before being notified of origin loss events, but would still receive immediate notification when a new origin appears.

5 Notification Delivery

Once a notification message is generated, it is delivered to the prefix owner's registered mailboxes through email. We choose email for delivery, since it is a ubiquitous delivery method on the Internet and uses TCP, which provides reliable data transfer. The email body is signed by the monitor to ensure its integrity. There are two types of messages: event-driven notifications and periodic refreshes. The event-driven notifications are triggered by origin set changes, and the email contains corresponding origin gains or losses. For example prefix 60.253.48.0/24, the notification messages look like the following:

<TYPE=gain, seqnum=1, GMT-TIME=20041221 12:52:33, PREFIX=60.253.48.0/24, NEW-SET={23918 31050 29257}, ORIGIN-GAINED=29257>

<TYPE=loss, seqnum=2, GMT-TIME=20041221 13:52:49, PREFIX=60.253.48.0/24, NEW-SET={29257 31050}, ORIGIN-LOST=23918>

The periodic notification is sent at fixed time interval (1 day by default), and the email contains the complete origin set at that moment. The periodic refresh message is a soft-state mechanism to provide additional system resilience against unforeseen errors. For instance, even if a notification is lost due to email server crash, the next refresh message will bring the owner's knowledge about the origin set up to date.

The major challenge in our system design is how to deliver notifications successfully even in the face of prefix hijacks. When a prefix is being hijacked, some data traffic on the Internet would go to the false origin instead of the true one. If the path from our server to the prefix owner is diverted to the false origin, then the owner would not receive the notification at the time when it is needed the most.

Due to the large scale of Internet routing, a prefix hijack is unlikely to affect all the paths towards the true origin. Thus in delivering the notification messages, our system uses multiple diversified paths to improve the chances of successful delivery. Ideally, we can send notifications from the monitors that still have path to the old origin. But this type of email forwarding service is not part of current BGP monitoring arrangement with commercial ISPs. Requiring email forwarding from monitors would undermine the deployability of our service. Thus we leave this as an option for future development,

[3]In other words, the penalty at time t is exactly one half of the penalty at time $(t - 2)$ hours, assuming no additional origin events were generated during that time

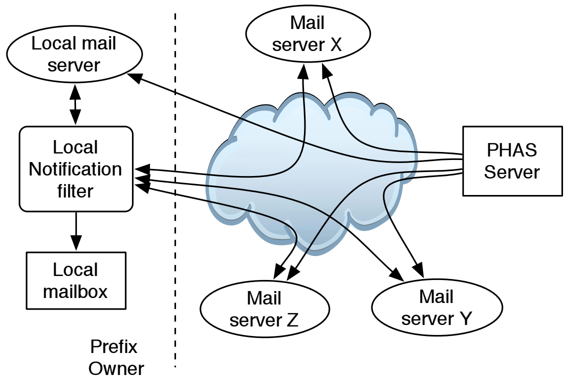


Figure 7: Notification setup

and instead ask prefix owners to take the responsibility of setting up multiple diversified delivery paths.

There are two practices recommended for prefix owners. First, when registering with our system, they should provide multiple email accounts on different servers that are also topologically diverse, for instance popular email services like GMail and Yahoo! mail. Secondly, they should have Internet access through multiple prefixes. ISPs often have several prefixes themselves, so this should not be a problem. For ones that only own a single prefix, a backup plan, such as a dial-up Internet access account, is recommended.

Figure 7 shows how the multiple diversified path delivery works. The owner of prefix P registers four email addresses, one within P, and three others, X, Y, and Z, in three different networks. Every notification message will be sent to all four mailboxes. The prefix owner's local filter will retrieve these four messages, and then process them. The email body will contain a sequence number, based on which the local filter decides whether it is a duplicate or is obsolete. Only emails with new contents pass through the filter and result in an alarm used for hijack detection. When a prefix P is hijacked, as long as the owner can access one of X, Y, Z, and our server, the notification will be delivered. Even if all four mailboxes are not accessible directly from the owner site, as long as the owner can access the Internet through another prefix, he/she can still retrieve the notification messages regarding the prefix P. The local filter also periodically polls the mailboxes. In the event that none of them is reachable, it is very likely that prefix owner's Internet access has problems, and the filter will generate an alarm to the operator. In summary, the combination of multiple topologically diversified mailboxes and multiple prefixes used for Internet access, ensures high delivery rate for notifications.

6 Local Notification Filter

PHAS does not associate a prefix with a true origin or false origin, and thus reports all origin set changes to the prefix owner. However, not all origin set changes may be of interest to the prefix owner, especially in the event that the origin set changes frequently. The local notification filter, is an important tuning block at the user side that enables the prefix owner to filter out unwanted alarms and alert the user for potential hijacks. In this section, we explain some basic building blocks for constructing filter rules and use examples to show how simple rules can control the notifications delivered to the user.

6.1 Constructing filtering rules

We define a rule to have the form "IF <condition> THEN <action>". There are two basic actions possible; ACCEPT results in the message being delivered and REJECT results in the message being dropped. The default action is ACCEPT, in case no rules are specified or no rules are fired. The local filter can contain various rules ordered by preference, and IF clauses can also be nested. While, multiple rules can be listed, for each notification message, an action of ACCEPT or REJECT can be performed only once. In other words, once an action is performed, no more rules are matched for that notification message. Hence, we encourage users to use rules that are simple and easy to understand and analyze.

To construct rules, we define the following constructs.

1. CONTAINS: defines what a particular key may contain.
2. DIFF: difference between sets.
3. LT, EQ, GT: correspond to the mathematical $<, =$ and $>$.
4. NOT: negates the construct it follows. E.g. one may use it with CONTAINS.
5. AND, OR: for combinations of conditions.
6. ANY and ALL: used to deal with sets in rules.

Examples

1. A rule specific to a prefix, and checking to see if the new origin is a known origin:

   IF <ORIGIN-GAINED EQ 29257 AND PREFIX EQ 60.253.48.0/24> THEN REJECT

2. A rule asking to drop all origin loss notifications:

   IF <TYPE EQ "loss"> THEN REJECT

Example of a bad Rule

1. A rule that checks for the existence of an AS in the ORIGIN-SET:

   IF <ORIGIN-SET CONTAINS 23918> THEN REJECT

In the event of a hijack that changes the origin set from $\{23918\}$ to $\{23918, X\}$, where X is the hijacker, the notification will not be delivered to the user, since the origin set still contains AS 23918.

6.2 Case Study

We now use a case study to show how simple rules can be used to deal with a real scenario. We choose prefix 60.253.48.0/24 as an example and look at the notifications from December 21, 2004 to December 28, 2004, when a known prefix hijack event happened. A sample of the notifications seen by the filter is shown below.

<TYPE=gain, GMT-TIME=20041221 04:44:45, PREFIX=60.253.48.0/24, NEW-SET={23918, 31050}, ORIGIN-GAINED=31050>

<TYPE=gain, GMT-TIME=20041221 12:52:33, PREFIX=60.253.48.0/24, NEW-SET={23918, 31050, 29257}, ORIGIN-GAINED=29257>

<TYPE=loss, GMT-TIME=20041221 13:52:49, PREFIX=60.253.48.0/24, NEW-SET={29257, 31050}, ORIGIN-LOST=23918>

<TYPE=loss, GMT-TIME=20041221 13:53:56, PREFIX=60.253.48.0/24, NEW-SET= {29257}, ORIGIN-LOST=31050>

For this prefix, we observed three origin ASes: AS 29257, AS 31050 and AS 23918. The origin set fluctuated between various combinations of these three ASes causing notifications to be sent to the owner. Without local filtering, all these legitimate changes would have resulted in alarms being sent to the prefix owner. However, the prefix owner, knowing all these three legitimate origin ASes, can set simple rules to filter out these changes:

IF <ORIGIN-GAINED EQ ANY {23918,31050,29257} > THEN REJECT

IF <ORIGIN-LOST EQ ANY {23918,31050,29257} > THEN REJECT

Note, each notification contains only one value for ORIGIN-GAINED or ORIGIN-LOST, and hence we can use EQ (equals) clause here. With this rule in place, the prefix owner would only receive an alarm when the origin changes passes both rules. Around 9:30 AM on Dec 24, 2004, such an alarm happened:

<TYPE=gain, GMT-TIME=20041224 09:30:29, PREFIX=60.253.48.0/24, NEW-SET={23918 9121}, ORIGIN-GAINED=9121>

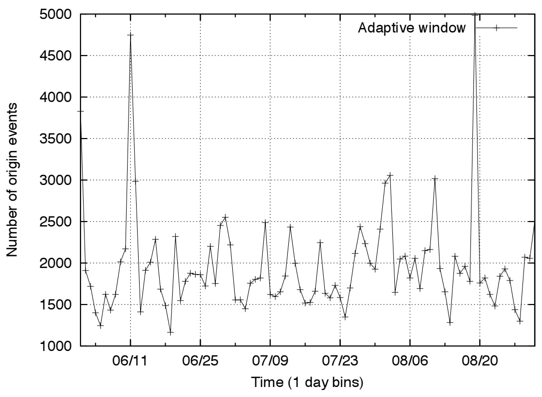


Figure 8: Origin events per day from June 1, 2005 to August 31, 2005

<TYPE=loss, GMT-TIME=20041224 11:35:02, PREFIX=60.253.48.0/24, NEW-SET= {23918}, ORIGIN-LOST=9121>

The first alarm indicates that AS 9121 is now hijacking the prefix 60.253.48.0/24. The owner knows that this is not a legitimate origin for this prefix, and can then take appropriate actions. An alarm is also generated to inform the owner that AS 9121 stopped announcing the prefix, indicating the matter has been resolved.

7 Evaluation

To evaluate the overhead of the system, we use BGP log data to calculate the number of origin events generated by the PHAS server, and the number of notifications received by each AS. We also apply our method to the data collected during known hijack events to show that PHAS can indeed catch those events. Finally, we use simulations to evaluate the success ratio of notification delivery using multi-path delivery scheme.

7.1 Notification Messages

Figures 8 and 9 plot the number of origin events per day over a 6 month period from June 2005 to November 2005. The origin events generated per day for month of December 2005 were shown in Figure 6 in Section 4. Throughout this period, we observed the number of events captured per day to be around 2000, with a few occasional spikes. From a system point of view, sending 2000 messages per day is manageable, even with multiple email delivery.

We now look from users' point of view to see how many notification messages they would receive if subscribed to receive PHAS alerts. We treat each origin event as one notification, assuming all prefixes are registered to receive alerts. For our evaluation, we use the

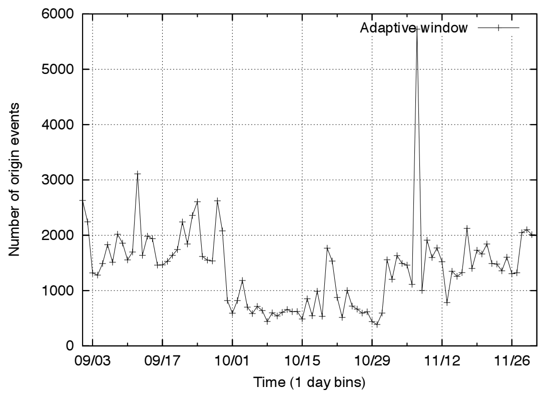


Figure 9: Origin events per day from September 1, 2005 to November 30, 2005

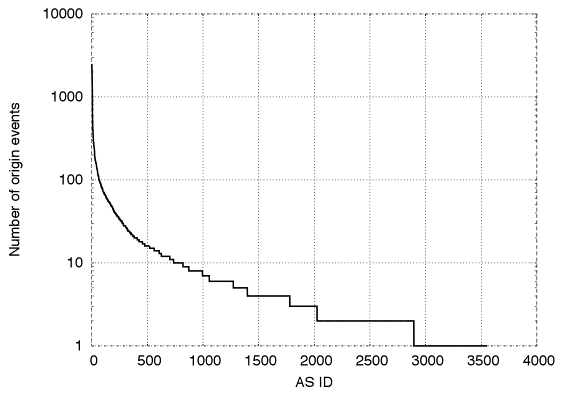


Figure 10: Distribution of events per AS for December 2005

events generated for the month of December 2005. We first evaluate the notifications received per prefix. From Figure 5 in Section 4, we see that only 20K out of more than 150K prefixes were involved in origin events. Of those 20K prefixes, almost all of them had less than 10 origin events per month. Only a handful of prefixes had more than 100 origin events per month. The worst case being 209.140.24.0/24 with 196 origin events. A closer look at the alarms revealed that the origin set alternated between $\{\}$ and $\{3043\}$, which indicates the prefix was unstable. From, these numbers for origin events, one can see that the number of notifications expected per prefix is quite small, except for some unstable prefixes. For cases of unstable prefixes, the owner's local filter will be able to handle such redundant notifications easily.

Since a prefix owner may register multiple prefixes, we also look at number of notifications expected per AS for the month of December 2005. For evaluation purpose, we estimated the prefixes registered by each AS by using the routing table to map every prefix to its origin AS. Figure 10 shows the number of origin events per AS for December 2005. Only about 3.5K ASes out of the total 18K ASes received notifications. Of those ASes that received notifications, 97% of them received less than 100 notifications in the entire month. The worst case was AS 29257, receiving 2501 notifications, with the $O_{SET}(P)$ fluctuating between combinations of 4 origins. These numbers for origin events per AS indicate that in most cases, an AS would receive a small number of notifications, and in extreme cases, local filters can once again deal with the common pattern of notifications. All of the above results show that the load of notification generation, transmission, and processing are easily manageable by a single machine, even when all the prefixes are registered with PHAS.

7.2 Detecting Known Events

We now check if our system would have caught some known prefix hijack events. One such prefix hijack occurred on May 7, 2005 when AS 174 hijacked one of Google's prefixes, 64.233.161.0/24, causing Google to be unreachable during this time. When run over this period of time, PHAS caught this origin set change and indicated AS 174 as the origin gained during this event.

A larger scale hijack event occurred on Dec 24, 2005. AS 9121 announced itself as origin to over 106K prefixes. PHAS detected 106082 unique prefixes with origin 9121 added to its origin set and a total of 217884 origin events. Most prefixes had 2 notifications, one reporting the addition of AS 9121, and the other reporting the removal of AS 9121.

Another case of hijack occurred on Jan 22, 2006, when AS 27506 announced itself as origin to some other's prefixes. For this day we detected 41 unique prefixes with AS 27506 as a new origin, and a total of 141 origin change events. For some prefixes, the AS 27506 was announced as origin, then withdrawn, and then reannounced and withdrawn again resulting in multiple origin events.

Overall, PHAS successfully caught every known prefix hijack due to false origin in a timely manner, and the timing matched reports from other sources.

7.3 Notification Delivery

To have multiple diverse paths for notification delivery, we recommend the prefix owners register multiple mailboxes and have multiple prefixes for Internet access. If they do have multiple prefixes, they can always receive the notification messages assuming only one is hijacked. In this subsection, we evaluate the effectiveness of using multiple mailboxes through simulations on Internet topology.

The approach is to take an Internet AS graph as the topology, tag each link with inferred relationship, assume the widely adopted "no-valley" routing policy on every node, then compute the shortest policy-compliant path between any two nodes. For each calculation, the input includes one true prefix origin, one false origin, and a set of mailboxes. Based on the computed shortest paths, we can find out the success ratio of notification delivery.

The AS Topology is collected from multiple sources, including BGP monitors, route servers, looking glasses, and routing registry [22]. The AS relationship is inferred using the method in [21]. Two set of mailboxes are used for comparison. The first set is RouteViews (AS 3582) only, which is called "direct delivery" without other mailbox. The second set is RouteViews plus GMail (AS 15169), Yahoo Mail (AS 10310), and Hotmail (AS 12076). We randomly picked 276 ASes to form the origin pairs. They are 15 tier-1 ASes, 21 tier-2 ASes, 20 tier-3 ASes, 20 tier-4 ASes, and 200 tier-5 ASes. We exhaust all the combinations of origin pairs, a total of 75900 cases.

Given an origin pair, some nodes will take the path to the true origin, while the others will take the path to the false origin. If a mailbox node takes the path to the true origin, the prefix owner will be able to access this mailbox and receive the notification. Otherwise the notification is lost. *Delivery ratio* is defined as the percentage of mailboxes that take the path to the true origin.

Note the simulation results will be symmetric. That is, suppose there is 20% delivery ratio for a given pair of true origin and a false origin, then it will be 80% when the role of these two origins switches. Since we exhaust all combinations of origin pairs, whenever there is a case of $a\%$ delivery ratio, there will be a corresponding case of $(1 - a\%)$ delivery ratio.

In our path computation, we use random tie-breaking when there're multiple shortest paths. For example, if a mailbox has two equal paths, one leads to the true origin, the other leads to the false origin, we count this as 0.5 notifications from this mailbox can be delivered.

Figure 11 compares the delivery ratio of with and without additional mailboxes. Without the three additional mailboxes, about 30% of notifications will be guaranteed delivered, about 30% of notifications will be lost for sure, and the rest may be delivered by certain probability. With the three additional mailboxes, the non-delivery ratio drops to about 10%.

Figure 12 shows the number (not the ratio) of notifications that can get delivered. In about two thirds ($x \geq 1$) of the cases, we have at least one messages are guaranteed to be delivered. It doubles compared with using only the direct delivery (30%). This suggests that three additional mailboxes can greatly improve the notification delivery, but we may still need more mailboxes for higher success ratio.

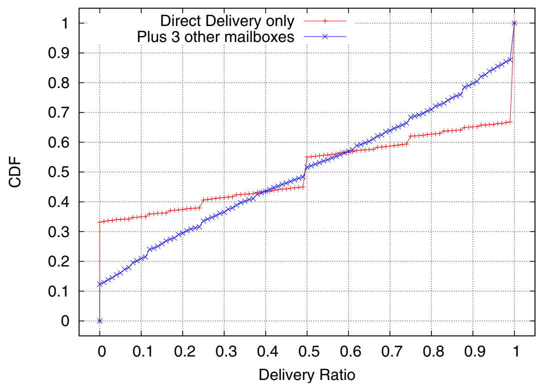


Figure 11: Delivery Ratio

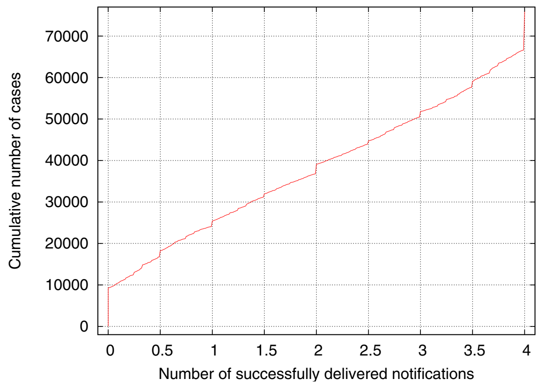


Figure 12: Delivery Number

8 Extensions to basic system

So far we have focused on detecting false origins. In this section, we discuss other ways of hijacking a prefix besides directly announcing a prefix and discuss extensions to the current system to deal with some of these cases.

8.1 Classification of Prefix hijack

At the highest level, the attacker AS could target a prefix that is already being announced by another AS, which we term as valid prefix. The attacker may pretend to be the owner of this prefix and originate the prefix resulting in a false origin hijack, that is the focus of this paper. Another way to hijack a prefix is by announcing a valid origin, but report invalid path to the origin. For false paths, we separate the case of *false last hop*, from false information on any other hop in the path, since the prefix

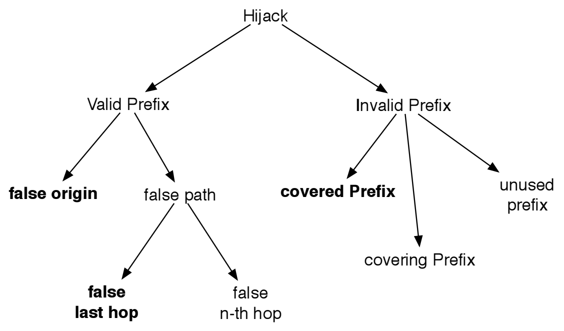


Figure 13: Types of prefix hijacks

owner's AS knows its immediate neighbors, and hence can identify whether the last hop is valid or not.

An attacker AS may also announce a prefix that is not being announced by another AS, termed as invalid prefix. If the attacker announces a sub-prefix of some valid prefix, termed as a *covered prefix hijack*, then routers in the Internet may contain routes to both the victim AS's prefix as well as the attacker's prefix. However, if the destination IP of a packet being routed, falls under the attacker's prefix space, then due to longest prefix match, the data would be forwarded to the attacker. An attacker AS may also announce a less specific prefix than a valid prefix, termed as a *covering prefix hijack* but will receive traffic, only when the route to the valid prefix is withdrawn. For example, if AS 110 announces 131.0.0.0/8, then AS A would route traffic destined to the valid prefix 131.179.0.0/16, to AS 110 only when the prefix 131.179.0.0/16 is withdrawn. Finally, an AS may announce an invalid prefix that does not conflict with any used prefix space. For example, spammers are known to use unused prefixes for spam purpose. Figure 13 shows the classification explained above.

Prefix hijacks could also include combinations of various types in Figure 13. E.g. AS 110 announcing 131.179.0.0/24 (invalid covered prefix) with the path $\{110, 52\}$ (invalid last hop). In Figure 13, the hijacks in bold (false origin, covered prefix, false last hop) are the ones where the prefix owner knows of what is legitimate and what may not be, and protection against these attacks is the focus of PHAS. We now discuss two other sets to deal with covered prefix hijack and false last hop hijack.

8.2 Sub-prefix Set

The idea of using a sub-prefix set is to provide the owner of an IP prefix with the information about whether anybody is announcing a more specific prefix under its assigned space. This would catch hijacking event where a prefix, say 131.179.96.0/26 is announced by a hijacker AS 100, but the prefix is part of the address space covered by 131.179.0.0/24, which is owned by AS 52.

For an IP prefix x, some or all of its assigned address space might get further divided into a number of longer prefixes. Each of these prefixes is a known as a covered prefix of x . The set of all covered prefixes of x observed from the BGP monitors, is denoted as $CP(x)$. For example, if UCLA announces 131.179.0.0/16 as well as 131.179.96.0/24 and 131.179.59.0/24, then $CP(131.179.0.0/16)$ is $\{131.179.96.0/24, 131.179.59.0/24\}$.

We define a sub-prefix set $SP_{SET}(x)$ to consist of all $y \in$ *CP(x)* such that there does not exist another prefix $z \in$ *CP(x)* with $y \in CP(z)$. In other words, the set $SP_{SET}(x)$ contains only the first level covered prefix for prefix x.

As an example of how this SP_{SET} could be useful, we present a case from Jan 22, 2006. The prefix 208.0.0.0/11, owned by Sprint, generated one origin event at 5:06 am UTC indicating that the sub-prefix set had changed from $\{\}$ to $\{208.28.1.0/24\}$ with origin $\{27506\}$. The prefix in question, 208.28.1.0/24 is not usually seen in the global routing tables, but in this case AS 27506 announced this prefix, which covers a portion of Sprint's 208.0.0.0/11 prefix space, thus resulting in a hijack.

8.3 Last Hop Set

The last hop set is maintained with the objective of detecting false last hops in BGP announcements. Once again, the owner of the prefix would know the legitimate next hops based on peering agreements and reports of such changes would allow the owner to detect false last hops in BGP paths.

We define an last hop set $LH_{SET}(A)$ as the set of last hops for all prefixes with AS A as the origin. For example, if M_1 observed a path (7018, 1239, 52) to prefix P_1, M_2 has a path (3356, 1239, 52) to P_2, and M_3's path to P_3 is (701, 852,52), then the last hop set of AS 52, or $LH_{SET}(52)$, is $\{1239, 852\}$. Note, that last hop is defined for an AS, and not for a prefix, since it is reflecting topological connectivity.

The main objective of using the sub-prefix set and the last hop set is to identify potential hijacks involving more specific address space and last hop changes. However, the sub-prefix set for large address blocks like 12.0.0.0/8 can be potentially huge, and may cause lots of dynamics. Similarly, the size of last hop sets for nodes with rich connectivity (e.g. tier 1 ISP) can also be significant, and may fluctuate a lot. For future work, we plan to understand the dynamics of these two sets, define how to use these sets, and include them as a part of the PHAS system.

9 Related Work

Various prefix hijack events have been reported to NANOG [10] mailing list from time to time. [23] and [8] studied the exact prefix hijack as part of the MOAS (Multiple Origin AS) problem, in which one prefix has multiple origin ASes in the routing table. These studies show that one prefix can be legitimately announced by multiple origin ASes, but can also be hijacked due to mis-configurations.

Existing proposals to address prefix hijack problem can be categorized into two types: cryptography based, and non-crypto based. Crypto-based solutions, such as [18], [1],[3], [11], [6], [17], require BGP routers to sign and verify the origin AS and the path, which have significant impact on router performance. Furthermore, these solutions are not easily deployable because they all need changes to router software, and some require public key infrastructures.

Non-crypto proposals include [2], [20], [24], and [5]. IRV approach in [2] lets each AS designate a server that answers queries regarding BGP security. [20] lets the router give preference to stable routes over transient ones which can be results of prefix hijacks. Similarly, in PG-BGP [5], a router detects prefix hijacks by monitoring the origin ASes in BGP announcements for each prefix over time. A transient origin AS of a prefix is considered as anomalous, and router avoids using the anomalous routes whenever possible. PG-BGP also detects covered prefix hijacks using similar approach. In [24], prefix owners attach additional information to the routing updates, so that remote routers could detect prefix hijacks. All the Above non-crypto proposals require changes to router softwares, router configurations, or the ways that operators run their networks.

Compared to all of the above proposals, the biggest advantage of our system is that it is fully deployable. PHAS can be up and running without requiring cooperation from multiple ISPs, registry authorities, router vendors, or even end users. While other approaches focus on detecting prefix hijack at remote ASes, we simply notify the prefix owner about the origin changes, thus allowing the prefix owners to detect prefix hijacks with a high accuracy.

Three other related works [19, 7, 9] are also fully deployable. [19] utilizes the data from RouteViews or RIPE and visualizes the origin AS changes of the prefixes for visual detection of the prefix hijacks. [7] proposes an alarming algorithm for prefix hijacks and path hijack, based on the the public BGP data, and the geographic information of the each AS from the *whois* database. The key observation is that if two edge ASes are connected to each other or legitimately originate the same prefixes, they are geographically close. Violation of this observation will trigger alarms.

The RIPE MyASN project [9] is probably the most similar service to ours, but its design is based on a fundamentally different philosophy. In the MyASN project, a prefix owner registers the valid origin set for a prefix. MyASN then tracks roughly the equivalent of our instantaneous origin set for this prefix. An alarm is triggered when any invalid origin AS appears. Our approach reports the origin set changes to the prefix owner, and any filtering or checking is done at the user site. This is a subtle difference, but has important implications.

First, filtering at the user side provides the greatest degree of flexibility to the detection algorithm. Users can apply any filtering criteria or detection algorithm on the data. When the filtering is done at the service site like MyASN, it is limited to what the service interface could provide. Obviously for security reason, the service site cannot allow arbitrary filtering script to be uploaded. If prefix owners cannot achieve their filtering goal at the service site, they have to deploy local filter anyway.

Second, it is critical for the server-based filtering to have the most up-to-date information needed for prefix hijack detection. The valid origin set must be updated at MyASN server whenever the prefix has a different origin set. It's especially hard to do update in face of an on-going prefix hijack. When a new hijack happens, the prefix owner may want to change the filtering rule, but is unable to do so due to the attack. Our approach does not does not suffer from this problem.

10 Conclusion

In this paper we described the design of PHAS, a Prefix Hijack Alert System. Rather than attempting an accurate route hijacking detection algorithm, PHAS aims at providing timely notification of origin AS changes to the owners of individual prefixes in a reliable way. The prefix owners can then easily identify real hijack alerts and filter out normal origin changes. By avoiding running complex data processing at BGP data collectors, PHAS can be quickly implemented and run with little overhead at the data collectors. By automating the email processing at the user end, PHAS provides network operators with realtime alerts to potential prefix hijacks while adding virtually no overhead to the operation tasks.

PHAS leverages on the existing routing logs for data input and the universally available email system for notification delivery. PHAS is light on authentication of users because its information is derived from publicly available data, and is light on data filtering because it simply provides information to users for hijack detection. As a result PHAS is light weight and readily deployable. As next step we plan to implement and install PHAS at RouteViews for trial deployment. We expect to gain further insight on how to improve PHAS through experience.

References

[1] W. Aiello, J. Ioannidis, and P. McDaniel. Origin Authentication in Interdomain Routing. In *Proceedings of 10th ACM Conference on Computer and Communications Security*, pages 165–178. ACM, October 2003. Washington, DC.

[2] G. Goodell, W. Aiello, T. Griffin, J. Ioannidis, P. McDaniel, and A. Rubin. Working around BGP: An incremental approach to improving security and accuracy of interdomain routing. In *NDSS*, 2003.

[3] Y.-C. Hu, A. Perrig, and M. Sirbu. SPV: Secure path vector routing for securing bgp. In *Proceedings of ACM Sigcomm*, August 2004.

[4] Geoff Huston. Auto-detecting hijacked prefixes? a presentation at RIPE-50, May 2005 http://www.potaroo.net/presentations/index.html.

[5] J. Karlin, S. Forrest, and J. Rexford. Pretty good bgp: Protecting bgp by cautiously selecting routes. Technical Report TR-CS-2005-37, University of New Mexico, Octber 2005.

[6] S. Kent, C. Lynn, and K. Seo. Secure border gateway protocol (S-BGP). *IEEE JSAC Special Issue on Network Security*, 2000.

[7] Christopher Kruegel, Darren Mutz, William Robertson, and Fredrik Valeur. Topology-based detection of anomalous bgp messages. In *6th Symposium on Recent Advances in Intrusion Detection (RAID)*, 2003.

[8] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *Proceedings of ACM Sigcomm*, August 2002.

[9] Ripe myasn system. http://www.ris.ripe.net/myasn.html.

[10] The NANOG Mailing List. http://www.merit.edu/mail.archives/nanog/.

[11] J. Ng. Extensions to BGP to Support Secure Origin BGP. ftp://ftp-eng.cisco.com/sobgp/drafts/draft-ng-sobgp-bgp-extensions-02.txt, April 2004.

[12] D. Pei, D. Massey, and L. Zhang. A Framework for Resilient Internet Routing Protocols. *IEEE Network Special Issue on Protection, Restoration, and Disaster Recovery*, 2004.

[13] Alin C. Popescu, Brian J. Premore, and Todd Underwood. Anatomy of a leak: As9121. http://www.nanog.org/mtg-0505/underwood.html.

[14] Anirudh Ramachandran and Nick Feamster. Understanding the network-level behavior of spammers. In *Proceedings of ACM SIGCOMMq*, 2006.

[15] Y. Rekhter and T. Li. Border Gateway Protocol 4. RFC 1771, SRI Network Information Center, July 1995.

[16] Y. Rekhter and T. Li. A Border Gateway Protocol (BGP-4). *Request for Comment (RFC): 1771*, 1995.

[17] B. R. Smith and J. J. Garcia-Luna-Aceves. Securing the border gateway routing protocol. In *Global Internet'96*, November 1996.

[18] L. Subramanian, V. Roth, I. Stoica, S. Shenker, and R. H. Katz. Listen and whisper: Security mechanisms for bgp. In *Proceedings of ACM NDSI 2004*, March 2004.

[19] S. T. Teoh, K.-L. Ma, S. F.Wu, D. Massey, X. Zhao, D. Pei, L. Wang, L. Zhang, and R. Bush. Visual-based anomaly detection for bgp origin as change (oasc) events. In *IFIP/IEEE DistributedSystems: Operations and Management (DSOM)*, pages 155–168, October 2003.

[20] L. Wang, X. Zhao, D. Pei, R. Bush, D. Massey, A. Mankin, S. Wu, and L. Zhang. Protecting BGP Routes to Top Level DNS Servers. In *Proceedings of the ICDCS 2003*, 2003.

[21] Jianhong Xia and Lixin Gao. On the evaluation of AS relationship inferences. In *Proc. of IEEE GLOBECOM*, December 2004.

[22] Beichuan Zhang, Raymond Liu, Daniel Massey, and Lixia Zhang. Collecting the internet as-level topology. *ACM SIGCOMM Computer Communications Review (CCR)*, 35(1):53–62, January 2005.

[23] X. Zhao, D. Pei, L. Wang, D. Massey, A. Mankin, S. Wu, and L. Zhang. An Analysis BGP Multiple Origin AS(MOAS) Conflicts. In *Proceedings of the ACM IMW2001*, Oct 2001.

[24] X. Zhao, D. Pei, L. Wang, D. Massey, A. Mankin, S. Wu, and L. Zhang. Dection of Invalid Routing Announcement in the Internet. In *Proceedings of the IEEE DSN 2002*, June 2002.

Passive Data Link Layer 802.11 Wireless Device Driver Fingerprinting

Jason Franklin[1] Damon McCoy[2] Parisa Tabriz[3] Vicentiu Neagoe[4]
Jamie Van Randwyk[5] Douglas Sicker[6]

Abstract

Motivated by the proliferation of wireless-enabled devices and the suspect nature of device driver code, we develop a passive fingerprinting technique that identifies the wireless device driver running on an IEEE 802.11 compliant device. This technique is valuable to an attacker wishing to conduct reconnaissance against a potential target so that he may launch a driver-specific exploit.

In particular, we develop a unique fingerprinting technique that accurately and efficiently identifies the wireless driver without modification to or cooperation from a wireless device. We perform an evaluation of this fingerprinting technique that shows it both quickly and accurately fingerprints wireless device drivers in real world wireless network conditions. Finally, we discuss ways to prevent fingerprinting that will aid in improving the security of wireless communication for devices that employ 802.11 networking.

1 Introduction

Device drivers are a primary source of security holes in modern operating systems [1]. Drivers experience error rates of three to seven times higher than other kernel code, making them the poorest quality code in most kernels [2]. There are a large number of different device drivers available, each being a potentially large body of code that is frequently modified to support new hardware features. These factors and the fact that drivers are often developed by programmers who lack intimate knowledge of the operating system kernel contribute to the disproportionately high number of bugs found in device drivers [3].

In general, device drivers execute in kernel space; hence, exploiting a vulnerable driver leads to compromise of the entire operating system. This threat is somewhat tempered by the fact that interacting with a driver typically requires physical access to a system. As a result, most security holes in device drivers are difficult to exploit remotely. For instance, it is hard to remotely interact with, much less exploit, a video or keyboard driver. Classes of drivers exist with which it is possible to interact without physical access to a system. Drivers for network devices such as wireless cards, Ethernet cards, and modems are examples. In particular, wireless network device drivers are easy to interact with and potentially exploit if the attacker is within transmission range of the wireless device. Today, the single most common and widespread wireless devices are those conforming to the IEEE 802.11 standards [4]. The vast number of 802.11 devices, the ease with which one may interact with their drivers, and the suspect nature of driver code in general has led us to evaluate the ability of an attacker to launch a driver-specific exploit by first fingerprinting the device driver.

Fingerprinting is a process by which a device or the software it is running is identified by its externally observable characteristics. In this paper, we design, implement, and evaluate a technique for fingerprinting IEEE 802.11a/b/g wireless network drivers. Our approach is based on statistical analysis of the rate at which common 802.11 data link layer frames are transmitted by a wireless device. Since most wireless exploits are dependent on the specific driver being used, wireless device driver fingerprinting can aid an attacker in launching a driver-specific exploit against a victim whose device is running a vulnerable driver.

Our technique is completely passive, meaning that a fingerprinter (attacker) needs only to be able to monitor wireless traffic from the fingerprintee (target, victim). This makes it possible for anyone within transmission range of a wireless device to fingerprint the device's wireless driver. Passive fingerprinting techniques have the advantage over active approaches in that they do not transmit data, making prevention of such techniques difficult. If an attacker can passively determine which driver a device is using, he can successfully gain information about his victim without fear of detection.

Our fingerprinting technique relies on the fact that most stations actively scan for access points to connect to by periodically sending out probe request frames. The algorithm used to scan for access points is not explicitly defined in the 802.11 standard. Therefore, it is up to the developers of device drivers to implement their own method for probing. This lack of an explicit specification for a probing algorithm in the 802.11 standard has led to the development of many wireless device drivers that perform this function entirely differently than other wireless device drivers. Our fingerprinting technique takes advantage of these implementation-dependent differences to accurately fingerprint a driver. Specifically, our method is based on statistical analysis of the inter-frame timing of transmitted probe requests. A timing-based approach has a number of advantages over a content-based approach. Primary among these is the fact that coarse-grained timing information is preserved despite the encryption of frame content as specified by security standards such as Wired Equivalent Privacy (WEP) or 802.11i [5].

Fingerprinting an 802.11 network interface card (NIC) is not a new concept. Many tools exist, such as Ethereal [6], that use the wireless device's Media Access Control (MAC) address to identify the card manufacturer and model number. A MAC address is an ostensibly unique character string that identifies a specific physical network interface. The IEEE Standards Association assigns each NIC manufacturer a special three-byte code, referred to as an Organizationally Unique Identifier (OUI), which identifies a particular manufacturer. While not part of the standard, most manufacturers use the next byte to specify the model of the NIC. There are a few notable advantages to using our method instead of relying on the information contained in the captured MAC address. First, the MAC address only identifies the model and manufacturer of the NIC. Our technique fingerprints the device driver (which resides at the operating system level), where the bulk of exploits rest. Second, some NICs can operate using multiple drivers, implying that the MAC address would not be enough information to identify what driver the NIC was using. Finally, whereas the MAC address is easily alterable in most operating systems, the features used by our passive technique are not a configurable option in any of the drivers tested.

Our testing demonstrates an accuracy for our method in identifying the driver that ranges from 77-96%, depending on the network setting. Our technique requires only a few minutes worth of network data to achieve this high level of accuracy. We also confirm that the technique can withstand realistic network conditions.

Contributions The main contributions of this paper is the design, implementation, and evaluation of a passive wireless device driver fingerprinting technique. Our technique is capable of passively identifying the wireless driver used by 802.11 wireless devices without specialized equipment and in realistic network conditions. In addition, we demonstrate that our technique is accurate, practical, fast, and requires little data to execute.

The remainder of the paper is organized as follows. Background material is presented in Section 2. Section 3 presents the design for our wireless device fingerprinting technique. Section 4 describes the implementation of our fingerprinting technique and Section 5 presents our experimental results and evaluation of our technique under realistic network conditions. Section 6 presents the limitations of our technique and Section 7 discusses possible ways to prevent driver fingerprinting. Finally, Section 8 examines related work and we conclude in Section 9.

2 Background: IEEE 802.11 Networks

Wireless technologies are encroaching upon the traditional realm of "fixed" or "wired" networks. The most widely adopted wireless networking technology thus far has been the 802.11 networking protocol, which consists of six modulation techniques, the most of common of which are the 802.11a, 802.11b, and 802.11g standard amendments. The price erosion and popularity of 802.11 capable hardware (especially 802.11b/g) has made wireless networks both affordable and easy to deploy in a number of settings, such as offices, homes, and wireless hot spots. Because of this, 802.11 is currently the most popular and common non-telephony communication protocol available for wireless communication [7].

The 802.11 standard defines a set of protocol requirements for a wireless MAC, or medium access control, which specifies the behavior of data link layer communication between stations in a wireless network. A station is simply a device with wireless capabilities, such as a laptop or PDA with a wireless networking interface. Throughout this paper, we often refer to stations as clients. Most 802.11 networks operate in infrastructure mode (as opposed to ad-hoc mode) and use an access point (AP) to manage all wireless communications; it is this type of network that is the setting for our fingerprinting technique. An example of a simple infrastructure network with three clients and one access point is depicted in Figure 1.

A key component of the 802.11 standard is the MAC specification that outlines the function of various communication frames. The MAC coordinates access to the wireless medium between stations and controls transmission of user data into the air via control and management frames. Higher-level protocol data, such as data produced by an application, is carried in data frames.

All 802.11 MAC frames include both a type and subtype field, which are used to distinguish between the three frame types (control, management, and data) and

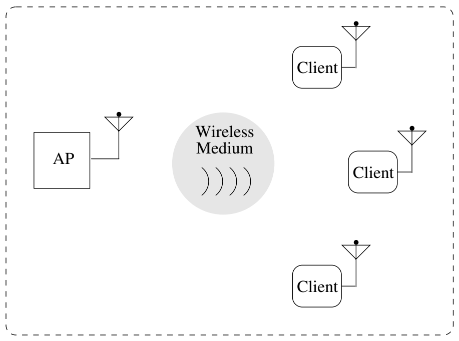


Figure 1: An infrastructure mode IEEE 802.11 network.

various subtypes. We consider only management frames in our passive fingerprinting technique, and specifically focus on probe request frames. Because of this, we only describe the most pertinent MAC frames communicated when a client joins a wireless network, and refer the reader to the IEEE 802.11 standard specification [4] for a more detailed description of MAC framing.

Each mobile client must identify and associate with an access point before it can receive network services. In a process called active scanning, clients use *probe request* frames to scan an area for a wireless access point, providing the data rates that the client can support inside fields of the probe request. If an access point is compatible with the client's data rates, it sends a probe response frame to acknowledge the request. Once a client identifies a network and authenticates to the access point via an authentication request and authentication response, the client can attempt to join the network by issuing an association request. If the association is successful, the access point will respond to the client with an association response that includes a unique association ID for future communications. At this point, all communication between a client and another machine, whether it resides within the wireless network or is located outside of it, is routed through and controlled by the access point.

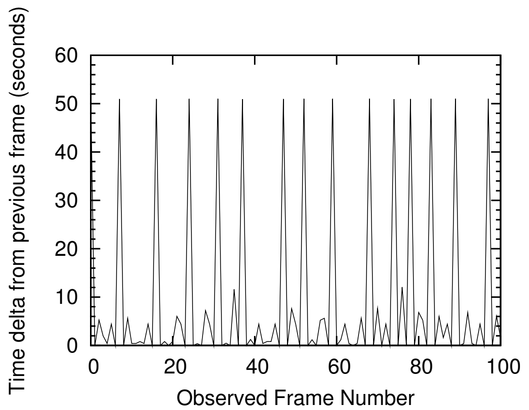
3 Fingerprinting Approach

Our fingerprinting technique is solely concerned with the active scan function in wireless clients. When actively scanning, clients send probe request frames to elicit responses from access points within transmission range. The IEEE 802.11 standard describes the active scan function of a client as follows. For each channel, the client broadcasts a probe request and starts a timer. If the timer reaches *MinChannelTime* and the channel is idle, the client scans the next channel. Otherwise, the client waits until the timer reaches *MaxChannelTime*, processes the received probe response frames and then scans the next channel. Further detailed specification of the active scanning function is not provided in the IEEE 802.11 standard. As a result, implementing active scanning within wireless drivers has become a poorly guided task. This has led to the development of many drivers that perform probing using slightly different techniques. By characterizing these implementation-dependent probing algorithms, we are able to passively identify the wireless driver employed by a device.

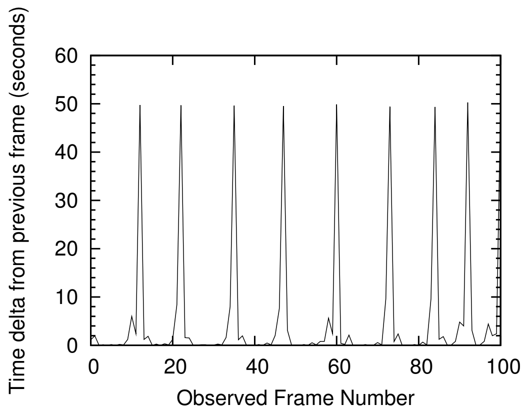
A number of factors affect the probing behavior of a client and make accurate fingerprinting without client cooperation a challenging task. From the perspective of an external fingerprinter, the probing behavior of a client is dependent on unobservable internal factors such as timers, and on uncontrollable external factors such as background traffic. A robust fingerprinting method cannot rely on client cooperation or assume a static environment, hence our technique uses machine learning to develop a model of a driver's behavior. This model is then used for future identification.

Having explained the intuition behind our technique, we turn our attention to two examples of representative probing behavior. Figure 2(a) and Figure 2(b) are plots of the time delta between arriving probe request frames as transmitted by two different wireless drivers. Both figures clearly depict a distinctly unique cyclic pattern. We further describe the pertinent features of Figure 2(b) as a way to characterize the differences between the probing patterns. Figure 2(b) is composed of a repeating pulse with an approximate amplitude of 50 seconds. These large pulses are occasionally preceded and/or followed by much smaller pulses ranging from 1-5 seconds. These pulses indicates that probing was occurring in bursts of probe request frames sent out, on average, every 50 seconds.

Upon closer inspection, one notices that the cyclic pattern exhibited by the driver probing is characterized by small variations. Our observations reveal there are two main reasons for this. The first reason is due to loss caused by signal interference. A fingerprinter could significantly reduce this type of loss by using a higher gain antenna found on commercial grade wireless cards. The second source of variation comes from wireless drivers continuously cycling through all eleven channels in the 2.4 GHz ISM band in search of other access points. The channel cycling can be considered an additional source of loss since probe request frames transmitted on unmonitored channels cannot be observed. Multiple wireless cards could be used to monitor all eleven channels simultaneously; however, we make the more realistic assumption that a fingerprinter has a single wireless card that can only monitor a small portion (e.g. one channel at any point in time) of the eleven channels. This loss indicates that some probe requests are missed, and statistical approaches are needed to compensate for the lost frames. Given the data described above, we character-



(a) D-Link driver for the D-Link DWL-G520 (802.11b/g) PCI wireless NIC



(b) Cisco driver for the Aironet AIR-CB21AG-A-K9 (802.11a/b/g) PCI wireless NIC

Figure 2: Plot of time delta from the previous arrival of probe request frames transmitted by two drivers.

ize the explicit probing behavior of a client by the sending rate of probe request frames. In the next section, we show how to leverage this characterization to accurately identify wireless drivers.

4 Device Driver Fingerprinting

The fingerprinting technique proceeds in two stages: trace capture and fingerprint generation. During trace capture, a fingerprinter within wireless transmission range of a fingerprintee captures 802.11 traffic, hereafter referred to as the trace. During fingerprint generation, the captured trace is analyzed using a supervised Bayesian approach to generate a robust device driver fingerprint.

4.1 Trace Capture

To begin the trace capture phase, we first consider how a fingerprinter might obtain a trace of probe request frames from a wireless device using widely available hardware and software. We assume a one-to-one mapping of MAC addresses to wireless devices, and believe this to be a reasonable assumption. Because each wireless NIC is assigned a unique MAC address by its manufacturer, the only cause for duplicate MACs on a network would be the result of a user reassigning his MAC address independently. However, as there are theoretically 2^{48} acceptable MAC addresses, the probability of a user choosing an existing MAC on the network is negligible[7]. In Section 7, we address the effects that violating this assumption has on our fingerprinting technique.

The fingerprinter can use any device that is capable of eavesdropping on the wireless frames transmitted by the fingerprintee. Therefore, the fingerprinter must be within receiving range of the fingerprintee's wireless transmissions. We assume the fingerprinter is using a single, high-gain, COTS (commercial off-the-shelf) wireless card. Next, the fingerprinter must configure their wireless card to operate in monitor mode; this mode allows the wireless card to capture frames promiscuously (e.g. whether they are specifically addressed to that wireless card or not). The fingerprinter must prevent their card from associating with an access point or sending its own probe request frames so collection is completely passive. This allows the fingerprinter to capture all frames sent on the current channel, including probe request frames, without interfering with the network's normal operation. We assume that the fingerprinter's machine is running an OS and driver combination that supports a wireless card in monitor mode. This can be easily done in Linux, FreeBSD, and Mac OS X. Finally, the fingerprinter can use a network protocol analyzer, such as Ethereal [6], to record the eavesdropped frames and filter out all irrelevant data. After following the above steps, the fingerprinter should have sufficient data to construct graphs similar to Figures 2(a) and 2(b).

4.2 Fingerprint Generation

After a trace has been captured, the data must be analyzed to characterize the probe request behavior. Previous work has shown that a simple supervised Bayesian approach is extremely accurate for many classification problems [8]. We chose to employ a binning approach to characterize the time deltas between probe requests because of the inherently noisy data due to frame loss.

Binning works by translating an interval of continuous data points into discrete bins. A bin is an internal value used in place of the true value of an attribute. The binning method smooths probabilities for the continuous attribute values by placing them into groups. Al-

Bin	Percentage	Mean
0	0.676	0.16
1.2	0.228	1.72
50	0.096	49.80

Table 1: Sample signature for the Cisco Aironet 802.11 a/b/g PCI driver

though binning causes some loss of information for continuous data, it allows for smooth probability estimates. Some noise is averaged out because each bin probability is an estimate for that interval, not individual continuous values. We chose to use equal-width binning where each bin represents an interval of the same size. While more sophisticated schemes may be available, this simple approach generated distinct fingerprints of probe inter-arrival times and provided a successful means for driver identification.

After performing a number of data analysis tests, we isolated two attributes from the probing rate that were essential to fingerprinting the wireless driver. The first attribute was the bin frequency of delta arrival time values between probe request frames. The second attribute was the average, for each bin, of all actual (non-rounded) delta arrival time values of the probe request frames placed in that bin. The first attribute characterizes the size of each bin and the second attribute characterizes the actual mean of each bin. Our next step was to create a signature (Bayesian model) for each individual wireless driver that embodies these attributes. Building models from tagged data sets is a common technique used in supervised Bayesian classifiers [9].

We now describe the process used to transform raw trace data into a device signature. To calculate the bin probabilities, we rounded the actual delta arrival time value to the closest discrete bin value. For example, if the bins were of a fixed width of size 1 second, any probe request frames with a delta arrival value in (0, 0.50] seconds would be placed in the 0 second bin, any probe request frames with a delta arrival value in (0.51, 1.50] seconds would be placed in the 1 second bin, and so forth. Based on empirical optimization experiments presented in our results section, we use an optimal bin width size of 0.8 seconds. The percentage of the total probe request frames placed in each bin is recorded along with the average, for each bin, of all actual (non-rounded) delta arrival time values of the probe request frames placed in that bin. These values comprise the signature for a wireless driver which we add to a master signature database containing all the tagged signatures that are created. An example of a signature created from the probe request frames in Figure 2(b) is shown in Table 1. New signatures can be inserted, modified, or deleted from the database without affecting other signatures. This allows collaborative signature sharing, similar to how Snort [10] intrusion detection signatures are currently shared.

Once the master signature database is created, a method is required to compute how "close" an untagged signature from a probe request trace is to each of the signatures in the master signature database.

4.3 Calculating Closeness

Let us now assume that a fingerprinter has obtained a trace and created a signature T of the probe request frames sent from the fingerprintee. Let p_n be the percentage of probe request frames in the nth bin of T and let m_n be the mean of all probe request frames in the nth bin. Let S be the set of all signatures in the master signature database and let s be a single signature within the set S. Let v_n be the percentage of probe request frames in the nth bin of s and let w_n be the mean of all probe request frames in the nth bin of s. The following equation was used to calculate the distance between the observed, untagged fingerprintee signature, T, and all known master signatures, assigning to C the distance value of the closest signature in the master database to T:

$$C = \min(\forall s \in S \sum_{0}^{n}(|p_n - v_n| + v_n|m_n - w_n|)) \quad (1)$$

Our technique iterates through all bins in T, summing the difference of the percentages and mean differences scaled by the percentage. The mean differences are scaled by the s bin percentage to prevent this value from dominating the bin percentage differences. We show in our results that the features included in a signature and our final method of calculating signature difference are effective in successfully fingerprinting wireless device drivers.

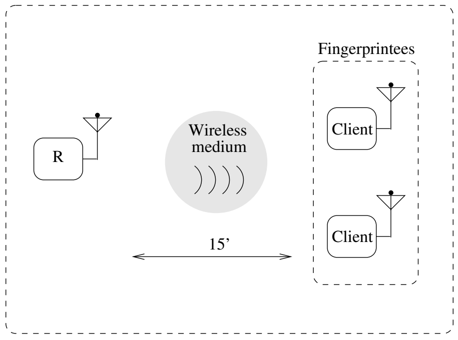
5 Evaluation

We tested our fingerprinting technique with a total of 17 different wireless interface drivers in their default configurations. We characterized wireless device drivers for the Linux 2.6 kernel, Windows XP Service Pack 1 and Service Pack 2, and Mac OS X 10.3.5. The machine we used to fingerprint other hosts' wireless drivers was a 2.4 GHz Pentium 4 desktop with a Cisco Aironet a/b/g PCI wireless card, running the Linux 2.6 kernel and the MadWifi wireless NIC driver [11]. Various Pentium III class desktop machines and one Apple PowerBook laptop were used as fingerprintee machines.

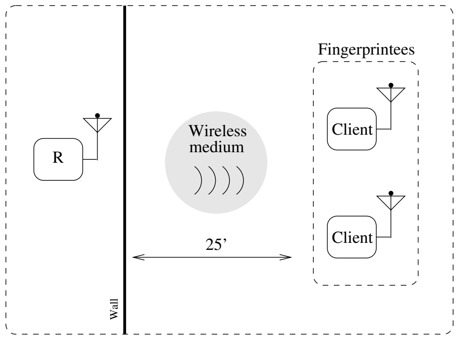
We address five primary characteristics that we expect any fingerprinting technique to be evaluated against. First, we investigate the resolution of our method. Specifically, we evaluate our identification granularity between drivers for different NICs, different drivers that support identical NICs, and different versions of the same driver. Second, we evaluate the consistency of our technique. We measure how successful our fingerprinting technique is in a variety of scenarios and over multiple network sessions, after operating system reboot, and when using the same driver to control different NICs. Third, we test the robustness of our technique. We conduct our experimentation in realistic network settings that experience loss rates similar to other wireless infrastructure networks. Fourth, we analyze the efficiency of our technique with respect to both data and time. Finally, we evaluate the resistance of our technique to varying configuration settings of a driver and evaluate the potential ways one might evade our fingerprinting technique.

To address these issues, we conducted a number of experiments using different wireless drivers and cards across a number of different operating system environments. In all cases, our technique successfully fingerprinted the wireless driver in at least one configuration. While the amount of time needed to collect the data varied across drivers and configurations, we required only a small amount of captured wireless traffic to fingerprint drivers accurately.

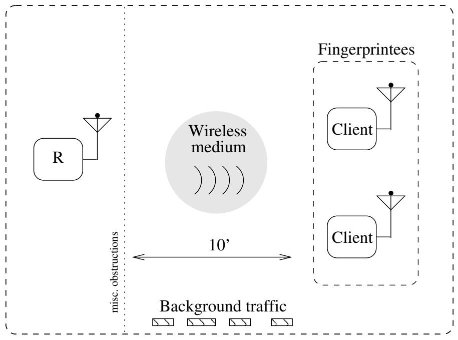
From our initial observations, we identified two properties of a device and driver that altered their signatures. The first property concerned whether the wireless device was unassociated or associated to an access point. Our initial experiments revealed that, by default, all wireless drivers transmit probe request frames when disassociated from an access point. Additionally, many continue to send probe requests even after association to an access point, though often not as frequently. The second property (only applicable to Windows drivers) concerns how the driver is managed. For many drivers, the Windows operating system can manage the configuration of the network settings for the wireless device instead of having a standalone (vendor provided) program perform those functions. The standalone program is provided by the manufacturer of the wireless device and often supports more configuration options for the specific driver, though also requires more user interaction to manage the device. We noticed slight differences in the behavior of probing depending on which option a user chose to manage their device. Due to these differences, we treated each of these property scenarios uniquely and created signatures to identify a driver under any of the appropriate cases.



(a) Test set 1 and master signature experimental setup.



(b) Test set 2 experimental setup.



(c) Test set 3 experimental setup.

Figure 3: Our test scenarios. R is the fingerprinter.

5.1 Building the Master Signatures

We collected trace data and constructed individual signatures with the same structure as the example signature in Table 1. This was repeated for all 17 wireless drivers in every configuration known to affect the signature and supported by the wireless driver. Drivers from Apple, Cisco, D-Link, Intel, Linksys, MadWifi (for Atheros chipset-based cards running under Linux), Netgear, Proxim, and SMC were included in our testing. A majority of the drivers included in our tests were for Windows; therefore most of the drivers initially had four individual signatures. We will refer to the four different configurations as follows: (1) unassociated and controlled by Windows, (2) unassociated and controlled by a standalone program, (3) associated and controlled by Windows, (4) associated and controlled by a standalone program. Three drivers did not support networking control by Windows (options 1 and 3), and four of the drivers tested did not transmit probe request frames when associated. This meant that initially, 57 signatures were compiled in the master signature database. We collected four signatures at a time and each signature trace contained a minimum of 12 hours worth of data points. A 30 minute portion of each trace was set aside and not used in signature training. This data was used as test set 1, which we further describe in the next section. As can be seen from Figure 3(a), the observing machine's antenna was placed approximately 15 feet from the fingerprintee machines, and no physical obstructions were present between the machines. Also, no 802.11 wireless traffic was detected besides the traffic generated by the fingerprintees.

After analyzing these signatures, we noted that changing configurations for some drivers had little impact on the probe request frame transmission rate and consequently, the generated signatures were indistinguishable from one another. We considered these signatures to be duplicates and removed all but one from the master signature database. This process could be automated by eliminating signatures that are insufficiently different from others with respect to some similarity threshold. There was only a single case where two of the drivers from the same manufacturer (Linksys) had indistinguishable signatures. For this case, we again left only a single signature in the master signature database. After pruning the database of all duplicate signatures, there remained 31 unique signatures. Each signature was tagged with the corresponding driver('s) name and configuration(s). The entire master signature database is included as Appendix A.

5.2 Collecting Test Data

We used the unused 30 minute trace from each of the 57 raw signature traces collected during master signature generation as test set 1. This scenario verifies that our signature generation adequately captures the probing behavior of the driver and that signatures can identify their associated drivers with a limited amount of traffic. To demonstrate that our technique is repeatable and still accurate in conditions other than where the signature data was originally collected, we repeated the 57 half hour experiments in two different physical locations. Using multiple environments helps to validate the consistency and robustness of our technique and suggests that it works well outside of lab settings. The arrangement for test set 2, as shown in Figure 3(b), was as follows: we placed the fingerprinter's antenna 25 feet from the fingerprintees with one uninsulated drywall placed in between the machines. As in Figure 3(a), no 802.11 wireless traffic was detected besides that generated by the fingerprintees. For test set 3, depicted in Figure 3(c), the observer's antenna was placed ten feet from the fingerprintees with two desks and other miscellaneous objects physically located between the machines. At this location, four to twelve other wireless devices were communicating during our data collection. Test set 2 might represent a wireless network in a semi-isolated setting, such as a hotel room with wireless access. Test set 3, on the other hand, represents a more congested wireless network, such as a network located in a coffee shop or airport.

Test Set	Successful	Total	Accuracy
1	55	57	96%
2	48	57	84%
3	44	57	77%

Table 2: Accuracy of fingerprinting technique by scenario.

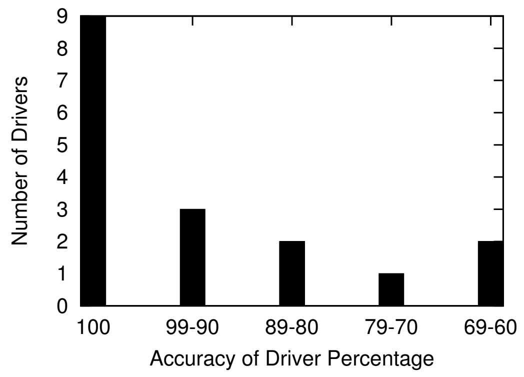


Figure 4: Number of individual drivers achieving an interval of accuracy over all test sets.

5.3 Fingerprinting Accuracy

The accuracy of our technique in correctly identifying the wireless driver operating a NIC for the three test scenarios is shown in Table 2. These results use the full half hour of data points. Later in this section, we will explore the effects of using less data points on the accuracy of our technique. The results also differed based on location. As expected, our technique is the most accurate for test set 1 (originally taken from the large signature traces) at 96%. The second most accurate test set was test set 2 (with only a single wall and no other 802.11 traffic) at 84%, and the last location had a 77% identification accuracy. These results indicate that different environments affect the accuracy of our technique. However, our technique remains reliable in all the the environments in which we tested.

Figure 4 demonstrates that our technique is perfectly

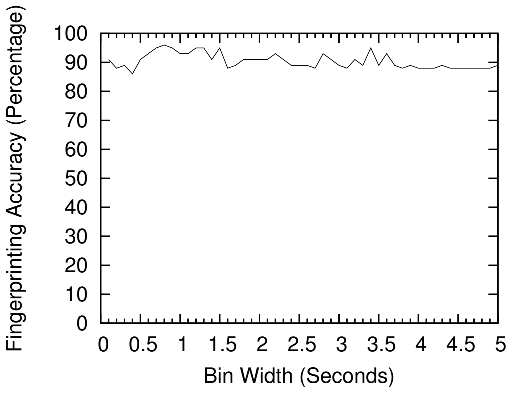


Figure 5: Empirical bin width tuning. Shows that 0.8 second wide bins generate the highest accuracy (96%) for test set 1.

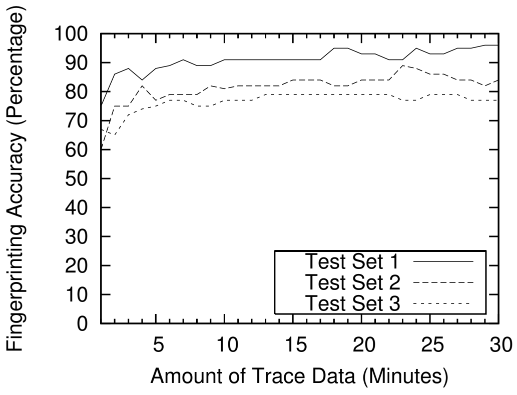


Figure 6: Effects of trace duration on fingerprinting accuracy.

accurate at fingerprinting nine of the wireless drivers and over 60% successful at identifying the other eight drivers. The accuracy of our method at identifying a particular driver is largely dependent on how dissimilar the driver's signature(s) are from other signatures in the master signature database. If the correct signature is similar to another in the database, noise such as background traffic may lead to our technique incorrectly fingerprinting a wireless driver. These results show that the majority of wireless drivers do have a distinct signature. It is important to note that even with drivers that have less unique fingerprints, we still correctly identify the driver for a majority of the test cases.

It is also relevant to note that in cases where the technique cannot uniquely identify a driver, it was able to narrow the possibilities down to those drivers that have similar signatures. Though not supported in the current implementation of our technique, it is conceivable to list the signatures in the master signature database that are close to the unidentified observed signature.

5.4 Empirical Bin Width Tuning

The bin width for signatures was empirically optimized during our experimentation on test set 1 by varying the size in testing and selecting an optimal width based on fingerprinting accuracy. This optimization began by starting with a bin width of 0.1 seconds and incrementally increasing the bin width by 0.1 seconds up to a bin width of 5.0 seconds. Figure 5 reveals that a bin width of 0.8 seconds produced the highest accuracy (96%) in test set 1, and thus, was the bin width used for the rest of our experiments.

5.5 Time Required to Fingerprint Driver

To address our technique's efficiency, we investigated the data and time thresholds required to accurately fingerprint a driver. Ideally, a fingerprinter would be able to identify a wireless driver in real time after only a small traffic trace. We measured the fingerprinting accuracy of our method in each test scenario with one minute of collected data and increased the amount of data in one minute increments until the full thirty minute trace from each setting was used. Figure 6 illustrates the accuracy of our technique in each of the three test cases corresponding to the amount of trace data used for fingerprinting.

Since the rate of probe request frames is different for most wireless drivers, it is difficult to estimate how many probe request frames will be recorded during one minute of observation, though for statistical interest, the average number of probes detected during one minute of observation was 10.79 across all of our testing scenarios. The accuracy of our technique is at least 60% in each of the three test cases after only one minute of traffic. These results show that our method successfully converges relatively fast on the correct wireless driver and needs only a small amount of communication traffic to do so.

6 Limitations

In the course of our evaluation, we discovered a few limitations of our fingerprinting technique. We discuss these in detail below.

6.1 Driver Versions

One of the original questions we posed concerned the resolution of our technique. We have shown that our technique is capable of distinguishing between different drivers the vast majority of the time. We are also interested in whether our method can distinguish between two different versions of the same wireless driver. A number of wireless card manufactures have released new versions of their wireless drivers to support new features. We tested our fingerprinting technique on six wireless drivers, with multiple driver versions available to determine if it was possible to distinguish between different versions of the same wireless driver. Our technique was unsuccessful in distinguishing between different versions of the same driver. This is a limitation of our fingerprinting technique since a new version of a driver might patch previous security vulnerabilities in the driver. However, even without the ability to distinguish between versions, our fingerprints greatly reduce the number of potential wireless drivers that a target system is running.

6.2 Hardware Abstraction Layer

Another unexpected limitation was found when testing the MadWifi driver for Linux. This driver works with most wireless cards containing the Atheros chipset because of the inclusion of a Hardware Abstraction Layer (HAL). This creates a more homogeneous driver environment since a majority of wireless cards currently available use the Atheros chipset. The side effect is that the lack of driver diversity reduces the appeal of fingerprinting wireless drivers. However, one drawback of a single (or relatively small number of) hardware abstraction layer(s) is that it magnifies any security vulnerability identified.

7 Preventing Fingerprinting

Several methods can be used to prevent our technique from successfully fingerprinting drivers. These methods include configurable probing, standardization, automatic generation of noise, driver code modification, MAC address masquerading, and driver vulnerability patching.

7.1 Configurable Probing

One solution to prevent our fingerprinting technique is for device drivers to provide the option to explicitly disable or enable probe request frames. It makes sense for this to be a configurable option not only to prevent fingerprinting but also to conserve power and bandwidth. Probe request frames are used to find networks matching the available data rates on the client device [7]. The SSID of the desired network can be specified or can be set to the broadcast SSID when probing for any available networks. By default, access points transmit beacon frames, which announce the access point's presence and some configuration information[8]. Thus, passively listening for beacons (i.e., turning off probe request frames) could be an effective method of discovering access points. Another solution would be to configure wireless device drivers, by default, to passively listen for beacons and only send probe requests for available networks when manually triggered by the user.

7.2 Standardization

An effective, but potentially difficult to implement solution for preventing driver fingerprinting is to specify the rate at which probe request frames are transmitted in a future IEEE standard for the 802.11 MAC. Another step towards standardization could result if a corporate body or open source consortium was formed to develop a standard agreed upon by all driver manufactures. If all driver manufactures adhered to such a standard, the described fingerprinting method would be rendered useless. Unfortunately, there are many obstacles preventing such a standard, the major factor being that some device manufacturers will not want to design devices that expend the power or bandwidth necessary to transmit probe requests at a standard rate. Due to this reason alone, it is doubtful that there will be any standardization agreed upon and followed by every driver manufacture concerning the rate of probe request frame transmission.

7.3 Automated Noise

Another strategy to prevent wireless driver fingerprinting is to generate noise in the form of cover probe request frames. Cover traffic disguises a driver by masking the driver's true rate of probe request transmission. Due to the fact that our technique uses statistical methods to filter out noise, the cover traffic would need to be sufficiently random and transmit enough cover to confuse our technique. A limitation of this approach is that the cover probe request frames waste bandwidth the device would otherwise use for wireless traffic, and for devices with limited power supplies, transmitting cover traffic would reduce battery life significantly. Also, given enough observation data, the fingerprinter might be able to filter away the noise and successfully fingerprint the driver. Generating noise is a difficult problem as many data mining algorithms have been shown to be effective in filtering out such noise and recovering the original data [12, 13, 14].

7.4 Driver Code Modification

For open source drivers such as the Madwifi drivers, the driver code could be modified to change the transmission rate of probe request frames. This alteration would fool our fingerprinting technique. However, this is only possible for open source drivers and would require a skilled programmer to alter the driver code. This would not be possible for many windows drivers, since most do not provide source code.

7.5 MAC Address Masquerading

Earlier, we made the assumption of a one-to-one mapping of MAC addresses to wireless devices. One method to prevent driver fingerprinting is to change the device's MAC address to match the MAC address of another device within transmission range. This would fool our fingerprinting technique into believing probe requests from two different wireless drivers are originating from the same wireless driver. There are a number of problems with this solution. First, the wireless device must make certain that the fingerprinter is within transmission range of both wireless devices. If the fingerprinter only observes probe request frames from one of the two devices, it will not be deceived. Also, since our method uses statistical methods to filter noise, the wireless device needs to make certain that the other device is transmitting enough probe request frames to mask its signature.

7.6 Driver Patching

While driver patching is not a full solution, we feel the creation of well thought out driver patching schemes would improve the overall security of device drivers as new driver exploits are found. Current research is being conducted to improve the process of patching security vulnerabilities [15, 16]. The device driver community should leverage this research to create more robust patching methods, and improve the overall level of driver security.

8 Related Work

Various techniques for system and device level fingerprinting have been used for both legitimate uses, such as forensics and intrusion detection, as well as malicious uses, such as attack reconnaissance and user profiling. The most common techniques take advantage of explicit content differences between system and application responses. Nmap [17], p0f [18], and Xprobe [19] are all open source, widely distributed tools that can remotely fingerprint an operating system by identifying unique responses from the TCP/IP networking stack. As the TCP/IP stack is tightly coupled to the operating system kernel, these tools match the content of machine responses to a database of OS specific response signatures. Nmap and Xprobe actively query the target system to invoke these potentially identifying responses. In addition to this active probing, p0f can passively fingerprint an operating system by monitoring network traffic from a target machine to some third party and matching characteristics of that traffic to a signature database. Data link layer content matching can also be used to identify wireless LAN discovery applications [20], which can be useful for wireless intrusion detection.

While datagram content identification methods are arguably the most simple, they are also limited to situations where datagram characteristics are uniquely identifiable across systems, as well as accessible to an outside party. Except for a few unique instances, 802.11 MAC-layer frame formatting and content is generally indistinguishable across wireless devices; because of this, more sophisticated methods are often required. In [21], the authors present a technique to identify network devices based on their unique analog signal characteristics. This fingerprinting technique is based on the premise that subtle differences in manufacturing and hardware components create unique signaling characteristics in digital devices. While the results of analog signal fingerprinting are significant, this method requires expensive hardware such as an analog to digital converter, IEEE 488 interface card, and digital sampling oscilloscope. Also, it is not clear from their analysis of wired Ethernet devices whether this method would be feasible in a typical wireless network setting where noise from both the environment and other devices is a more pressing consideration.

A device's clock skew is also a target for fingerprinting. A technique presented in [22] uses slight drifts in a device's TCP option clock to identify a network device over the Internet via its unique clock skew. Whereas our technique fingerprints which driver a wireless device is running, time skew fingerprinting is used to identify distinct devices on the Internet. Concerning security, unique device fingerprinting is often not as useful as driver and other types of software fingerprinting. As opposed to content based fingerprinting, both analog signal and time skew fingerprinting exploit characteristics of the underlying system hardware, making these techniques much more difficult to spoof.

Identification via statistical timing analysis in the context of communication patterns and data content has been especially studied in the area of privacy enhancing technologies. While network security mechanisms such as encryption are often utilized to protect user privacy, traffic analysis of encrypted traffic has proven successful in linking communication initiators and recipients participating in anonymous networking systems [23, 24]. Traffic analysis has also been applied to Web page fingerprinting. In [25], the authors demonstrate a technique that characterizes the inter-arrival time and datagram sizes of web requests for certain popular web sites. Using these web page characterizations, one can identify which sites users on wireless LANs are visiting despite these users browsing the Internet via encrypted HTTP traffic streams.

The techniques described above serve as only a survey of existing fingerprinting techniques for systems, devices, and even static content. The approaches vary from exploiting content anomalies in the TCP/IP stack to characterizing time-based system behavior at both the physical and software layers of a system. While the approaches vary, these contributions bring to light the true feasibility of fingerprinting via avenues otherwise assumed to be uniformly implemented across systems.

9 Conclusion

We designed, implemented, and evaluated a technique for passive wireless device driver fingerprinting that exploits the fact that most IEEE 802.11a/b/g wireless drivers have implemented different active scanning algorithms. We evaluated our technique and demonstrated that it is capable of accurately identifying the wireless driver used by 802.11 wireless devices without specialized equipment and in realistic network conditions. Through an extensive evaluation including 17 wireless drivers, we demonstrated that our method is effective in fingerprinting a wide variety of wireless drivers currently on the market. Finally, we discussed ways to prevent fingerprinting that we hope will aid in improving the security of wireless communication for devices that employ 802.11 networking.

10 Acknowledgments

Some of this work was performed while the authors were at Sandia National Laboratories - California. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000. The authors greatly appreciate the staff of Sandia National Laboratories for their assistance. The authors would like to thank John Bethencourt, Nikita Borisov, Frank Hemingway, Adam Lee, Kristen Pelon, Amanda Stephano, and the anonymous reviewers for their useful suggestions. This work was partially supported by NSF Grant ITR-0428887 (Spectrum Management Toward Spectrum Plenty) and the University of Colorado. Jason Franklin performed this research while on appointment as a U.S. Department of Homeland Security (DHS) Fellow. The views expressed in this paper do not necessarily reflect the policies and views of DHS, DOE, or affiliated organizations.

References

[1] Ken Ashcraft and Dawson R. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2002.

[2] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An Empirical Study of Operating System Errors. In *Proceedings of Symposium on Operating Systems Principles (SOSP 2001)*, October 2001.

[3] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proceedings of Symposium on Operating Systems Principles (SOSP 2003)*, October 2003.

[4] IEEE-SA Standards Board. IEEE Std IEEE 802.11-1999 Information Technology - Wireless LAN Medium Access Control (MAC) And Physical Layer (PHY) Specifications. IEEE Computer Society, 1999.

[5] IEEE-SA Standards Board. Amendment 6: Medium Access Control (MAC) Security Enhancements. IEEE Computer Society, April 2004.

[6] Ethereal: A network protocol analyzer. Web site, 2006. ⟨`http://www.ethereal.com`⟩.

[7] Matthew S. Gast. *802.11 Wireless Networks: The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2nd edition, 2005.

[8] Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian Network Classifiers. *Machine Learning*, 29(2-3):131–163, 1997.

[9] T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning*. Springer, 2001.

[10] Snort Intrusion Detection and Prevention system. Web site, 2006. ⟨`http://www.snort.org/`⟩.

[11] Madwifi: Atheros chip set drivers. Web site, 2006. ⟨`http://sourceforge.net/projects/madwifi/`⟩.

[12] D. Agrawal and C. C. Aggarwal. On the Design and Quantification of Privacy Preserving Data Mining Algorithms. In *Proceedings of Symposium on Principles of Database Systems*, 2001.

[13] R. Agrawal and R. Srikant. Privacy-preserving data mining. In *Proceedings of ACM SIGMOD*, May 2000.

[14] B. Hoh and M. Gruteser. Location Privacy Through Path Confusion. In *Proceedings of IEEE/CreateNet International Conference on Security and Privacy for Emerging Areas in Communication Networks (SecureComm 2005)*, 2005.

[15] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. OPUS: Online Patches and Updates for Security. In *Proceedings of 14th USENIX Security Symposium*, Aug 2005.

[16] John Dunagan, Roussi Roussev, Brad Daniels, Aaron Johnson, Chad Verbowski, and Yi-Min Wang. Towards a Self-Managing Software Patching Process Using Black-Box Persistent-State Manifests. In *First International Conference on Autonomic Computing (ICAC'04)*, 2004.

[17] Nmap: a free network mapping and security scanning tool. Web site, 2006. ⟨`http://www.insecure.org/nmap/`⟩.

[18] Project details for p0f. Web site, 2004. ⟨`http://freshmeat.net/projects/p0f/`⟩.

[19] Arkin and Yarochkin. Xprobe project page. Web site, August 2002. ⟨`http://sourceforge.net/projects/xprobe`⟩.

[20] Joshua Wright. Layer 2 Analysis of WLAN Discovery Applications for Intrusion Detection. Web site, 2002. ⟨`http://www.polarcove.com/whitepapers/layer2.pdf`⟩.

[21] Ryan Gerdes, Thomas Daniels, Mani Mina, and Steve Russell. Device Identification via Analog Signal Fingerprinting: A Matched Filter Approach. In *Proceedings of the Network and Distributed System Security Symposium Conference (NDSS 2006)*, 2006.

[22] Tadayoshi Kohno, Andre Broido, and K. C. Claffy. Remote Physical Device Fingerprinting. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (SP 2005)*, Washington, DC, USA, 2005.

[23] Jean-François Raymond. Traffic Analysis: Protocols, Attacks, Design Issues, and Open Problems. In *Proceedings of Privacy Enhancing Technologies Workshop (PET 2000)*, May 2000.

[24] Mathewson and Dingledine. Practical Traffic Analysis: Extending and Resisting Statistical Disclosure. In *Proceedings of Privacy Enhancing Technologies Workshop (PET 2004)*, May 2004.

[25] George Dean Bissias, Marc Liberatore, and Brian Neil Levine. Privacy Vulnerabilities in Encrypted HTTP Streams. In *Proceedings of Privacy Enhancing Technologies Workshop (PET 2005)*, May 2005.

[26] Mike Kershaw. Kismet. Web site, 2006. ⟨`http://www.kismetwireless.net/`⟩.

Appendix A

This appendix includes the entire master signature database from our evaluation section. It is organized with the name of the wireless driver, if the driver was associated (assoc) or unassociated (unassoc), and if Windows (win) was configuring the wireless device, or a standalone program (native). The values after the driver name and configuration are a set of tuples ordered as follows: (Bin Value, Percentage, Bin Mean Value).

cisco-abg-assoc-native (0.8,0.101,0.677)(1.6,0.108,1.450)(2.4,0.168,2.377)(3.2,0.021,2.928)(4,0.024,3.798)(4.8,0.028,4.691)(5.6,0.048,5.536)(6.4,0.034,6.303)(7.2,0.080,7.132)(8,0.032,7.830)(8.8,0.017,8.473)(9.6,0.044,9.607)(53.6,0.288,53.399)

cisco-abg-unassoc-native (0,0.514,0.048)(0.8,0.285,0.749)(1.6,0.037,1.656)(2.4,0.028,2.373)(3.2,0.067,3.264)(4.8,0.041,4.981)(5.6,0.025,5.521)

cisco-abg-unassoc-win (0,0.466,0.072)(0.8,0.126,0.720)(1.6,0.115,1.479)(2.4,0.056,2.345)(3.2,0.040,3.089)(4,0.025,3.843)(4.8,0.020,4.592)(5.6,0.019,5.415)(6.4,0.012,6.129)(8.8,0.013,8.554)(49.6,0.063,49.639)(50.4,0.026,50.146)

dwl-ag530-assoc-native (0,0.420,0.032)(0.8,0.108,0.590)(1.6,0.043,1.358)(2.4,0.011,2.067)(4.8,0.113,4.470)(5.6,0.060,5.477)(6.4,0.039,6.192)(7.2,0.030,7.206)(8,0.011,7.630)(12,0.010,11.829)(51.2,0.144,50.995)

dwl-ag530-unassoc-native (0,0.544,0.034)(0.8,0.052,0.597)(1.6,0.198,1.670)(6.4,0.053,6.659)(7.2,0.129,7.248)(8,0.012,7.806)

dwl-ag650-assoc-win (0,0.392,0.008)(0.8,0.231,0.549)(1.6,0.049,1.481)(2.4,0.030,2.416)(3.2,0.045,3.250)(4,0.067,4.092)(4.8,0.021,4.687)(58.4,0.164,58.198)

dwl-ag650-unassoc-win (0,0.606,0.084)(0.8,0.233,0.621)(1.6,0.090,1.689)(2.4,0.068,2.322)

dwl-g520-unassoc-native (0,0.533,0.054)(0.8,0.246,0.674)(1.6,0.072,1.541)(2.4,0.035,2.539)(3.2,0.079,2.989)(4,0.026,3.706)

dwl-g520-unassoc-win (0,0.527,0.055)(0.8,0.236,0.666)(1.6,0.134,1.523)(2.4,0.039,2.401)(3.2,0.044,3.109)(4,0.015,3.791)

engenuis-unassoc-win (0,0.193,0.059)(0.8,0.104,1.188)(1.6,0.609,1.271)(2.4,0.082,2.529)(4,0.011,3.814)

intel2100-assoc-win (0,0.766,0.019)(63.2,0.234,62.949)

intel2100-unassoc-win (0,0.927,0.055)(30.4,0.073,30.132)

intel-2200-assoc-native (0,0.591,0.107)(0.8,0.071,0.955)(1.6,0.079,1.495)(2.4,0.107,2.182)(120,0.050,120.254)(120.8,0.091,120.698)

intel-2200-unassoc-native (0,0.659,0.078)(0.8,0.015,0.882)(32.8,0.031,33.063)(34.4,0.139,34.765) (35.2,0.142,34.853)

intel-2915-assoc-native (0,0.659,0.080)(0.8,0.032,0.938)(1.6,0.037,1.426)(118.4,0.171,118.155) (119.2,0.076,119.193)

intel-2915-unassoc-native (0,0.668,0.083)(32.8,0.331,32.868)

linksys-pci-unassoc-win (0,0.348,0.165)(0.8,0.273,0.923)(1.6,0.032,1.262)(61.6,0.262,61.787)(62.4,0.027,62.270)(63.2,0.054,62.953)

madwifi-unassoc (72.8,0.881,72.988)(133.6,0.119,133.978)

netgear-assoc-win (0,0.423,0.001)(0.8,0.203,0.611)(1.6,0.038,1.552)(2.4,0.058,2.240)(3.2,0.037,3.206)(4,0.016,4.006)(4.8,0.060,4.731)(5.6,0.010,5.505)(57.6,0.149,57.498)

netgear-unassoc-win (0,0.560,0.061)(0.8,0.135,0.652)(1.6,0.077,1.532)(2.4,0.018,2.340)(3.2,0.023,3.125)(4,0.106,4.035)(4.8,0.071,4.566)

osx-airportb-unassoc (0,0.639,0.022)(10.4,0.361,10.295)

proxim-assoc-native (0,0.035,0.396)(0.8,0.377,0.585)(1.6,0.133,1.376)(2.4,0.016,2.078)(4.8,0.168,4.523)(5.6,0.035,5.535)(55.2,0.087,55.400)(56,0.024,56.017)(56.8,0.084,56.836)(57.6,0.022,57.435)

proxim-assoc-win (0,0.039,0.385)(0.8,0.329,0.585)(1.6,0.118,1.385)(2.4,0.020,2.055)(4.8,0.089,4.681)(5.6,0.089,5.500)(6.4,0.032,6.242)(7.2,0.013,7.167)(55.2,0.122,55.402)(56,0.036,56.037)(56.8,0.068,56.773)(57.6,0.012,57.466)

proxim-unassoc-win (0,0.540,0.052)(0.8,0.229,0.660)(1.6,0.090,1.555)(2.4,0.012,2.328)(4.8,0.055,5.011)(6.4,0.040,6.479)

smc-2532w-assoc-native (0,0.619,0.140)(0.8,0.028,0.477)(1.6,0.013,1.812)(60.8,0.013,60.907)(62.4,0.183,62.595)(63.2,0.118,62.899)

smc-2532w-unassoc-win (0,0.065,0.140)(0.8,0.047,0.727)(1.6,0.880,1.681)

smc-2632w-unassoc-native (0,0.511,0.083)(10.4,0.470,10.555)

smc-wpci-assoc-native (0,0.461,0.001)(0.8,0.117,0.588)(1.6,0.139,1.678)(2.4,0.020,2.185)(4,0.021,3.915)(4.8,0.093,5.028)(56,0.127,55.708)

smc-wpci-unassoc-native (0,0.563,0.038)(0.8,0.144,0.689)(1.6,0.014,1.790)(4,0.093,3.857)(4.8,0.079,4.952)(5.6,0.057,5.935)(6.4,0.026,6.178)

wpc54g-assoc-win (0,0.633,0.038)(0.8,0.114,0.437)(62.4,0.148,62.550)(63.2,0.105,62.981)

wpc54g-unassoc-native (0,0.623,0.054)(0.8,0.151,0.633)(62.4,0.172,62.299)(63.2,0.055,62.960)

Notes

[1]Carnegie Mellon University,`jfrankli@cs.cmu.edu`

[2]University of Colorado, Boulder, `damon.mccoy@colorado.edu`

[3]University of Illinois, Urbana-Champaign, `tabriz@uiuc.edu`

[4]University of California, Davis,`vneagoe@ucdavis.edu`

[5]Sandia National Laboratories,`jvanran@sandia.gov`

[6]University of Colorado, Boulder,`douglas.sicker@colorado.edu`

[7]It is important to note that some attackers will sniff the MAC addresses of other users on a wireless network to use as their own, giving them the ability to steal a connection or hide their malicious actions. Although we acknowledge that this scenario would bring about duplicate MAC addresses on a network, we believe it is far from the common case in most network settings.

[8]This is in contrast to disabling the SSID broadcast function. Disabling SSID broadcast simply forces an AP to send a string of spaces or a null string in the SSID field of the beacon frame. Kismet [26] reports this SSID as `<no ssid>`.

Static Detection of Security Vulnerabilities in Scripting Languages

Yichen Xie *Alex Aiken*
Computer Science Department
Stanford University
Stanford, CA 94305
{*yxie,aiken*}*@cs.stanford.edu*

Abstract

We present a static analysis algorithm for detecting security vulnerabilities in PHP, a popular server-side scripting language for building web applications. Our analysis employs a novel three-tier architecture to capture information at decreasing levels of granularity at the intrablock, intraprocedural, and interprocedural level. This architecture enables us to handle dynamic features of scripting languages that have not been adequately addressed by previous techniques.

We demonstrate the effectiveness of our approach on six popular open source PHP code bases, finding 105 previously unknown security vulnerabilities, most of which we believe are remotely exploitable.

1 Introduction

Web-based applications have proliferated rapidly in recent years and have become the *de facto* standard for delivering online services ranging from discussion forums to security sensitive areas such as banking and retailing. As such, security vulnerabilities in these applications represent an increasing threat to both the providers and the users of such services. During the second half of 2004, Symantec cataloged 670 vulnerabilities affecting web applications, an 81% increase over the same period in 2003 [17]. This trend is likely to continue for the foreseeable future.

According to the same report, these vulnerabilities are typically caused by programming errors in input validation and improper handling of submitted requests [17]. Since vulnerabilities are usually deeply embedded in the program logic, traditional network-level defense (e.g., firewalls) does not offer adequate protection against such attacks. Testing is also largely ineffective because attackers typically use the least expected input to exploit these vulnerabilities and compromise the system.

A natural alternative is to find these errors using static analysis. This approach has been explored in WebSSARI [7] and by Minamide [10]. WebSSARI has been used to find a number of security vulnerabilities in PHP scripts, but has a large number of false positives and negatives due to its intraprocedural type-based analysis. Minamide's system checks syntactic correctness of HTML output from PHP scripts and does not seem to be effective for finding security vulnerabilities. The main message of this paper is that analysis of scripting languages need not be significantly more difficult than analysis of conventional languages. While a scripting language stresses different aspects of static analysis, an analysis suitably designed to address the important aspects of scripting languages can identify many serious vulnerabilities in scripts reliably and with a high degree of automation. Given the importance of scripting in real world applications, we believe there is an opportunity for static analysis to have a significant impact in this new domain.

In this paper, we apply static analysis to finding security vulnerabilities in PHP, a server-side scripting language that has become one of the most widely adopted platforms for developing web applications.[1] Our goal is a bug detection tool that automatically finds serious vulnerabilities with high confidence. This work, however, does not aim to verify the absence of bugs.

This paper makes the following contributions:

- We present an interprocedural static analysis algorithm for PHP. A language as dynamic as PHP presents unique challenges for static analysis: language constructs (e.g., `include`) that allow dynamic inclusion of program code, variables whose types change during execution, operations with semantics that depend on the runtime types of the operands (e.g., $<$), and pervasive use of hash tables and regular expression matching are just some features that must be modeled well to produce useful results.

[1] Installed on over 23 million Internet domains [14], and is ranked fourth on the TIOBE programming community index [18].

To faithfully model program behavior in such a language, we use a three-tier analysis that captures information at decreasing levels of granularity at the intrablock, intraprocedural, and interprocedural levels. This architecture allows the analysis to be precise where it matters the most–at the intrablock and, to a lesser extent, the intraprocedural levels–and use agressive abstraction at the natural abstraction boundary along function calls to achieve scalability. We use symbolic execution to model dynamic features inside basic blocks and use block summaries to hide that complexity from intra- and inter-procedural analysis. We believe the same techniques can be applied easily to other scripting languages (e.g., Perl).

- We show how to use our static analysis algorithm to find SQL injection vulnerabilities. Once configured, the analysis is fully automatic. Although we focus on SQL injections in this work, the same techniques can be applied to detecting other vulnerabilities such as cross site scripting (XSS) and code injection in web applications.
- We experimentally validate our approach by implementing the analysis algorithm and running it on six popular web applications written in PHP, finding 105 previously unknown security vulnerabilities. We analyzed two reported vulnerabilities in PHP-fusion, a mature, widely deployed content management system, and construct exploits for both that allow an attacker to control or damage the system.[2]

The rest of the paper is organized as follows. We start with a brief introduction to PHP and show examples of SQL vulnerabilities in web application code (Section 2). We then present our analysis algorithm and show how we use it to find SQL injection vulnerabilities (Section 3). Section 4 describes the implementation, experimental results, and two case studies of exploitable vulnerabilities in PHP-fusion. Section 5 discusses related work and Section 6 concludes.

2 Background

This section briefly introduces the PHP language and shows examples of SQL injection vulnerabilities in PHP.

PHP was created a decade ago by Rasmus Lerdorf as a simple set of Perl scripts for tracking accesses to his online resume. It has since evolved into one of the most popular server-side scripting languages for building web applications. According to a recent Security Space survey, PHP is installed on 44.6% of Apache web servers [16], adopted by millions of developers, and used or supported by Yahoo, IBM, Oracle, and SAP, among others [14].

Although the PHP language has undergone two major re-designs over the past decade, it retains a Perl-like syntax and dynamic (interpreted) nature, which contributes to its most frequently claimed advantage of being simple and flexible.

PHP has a suite of programming constructs and special operations that ease web development. We give three examples:

1. **Natural integration with SQL:** PHP provides nearly native support for database operations. For example, using inline variables in strings, most SQL queries can be concisely expressed with a simple function call

```
$rows=mysql_query("UPDATE users SET
    pass='$pass' WHERE userid='$userid'");
```

   Contrast this code with Java, where a database is typically accessed through *prepared statements*: one creates a statement template and fills in the values (along with their types) using *bind variables*:

```
PreparedStatement s = con.prepareStatement
  ("UPDATE users SET pass = ?
    WHERE userid = ?");
s.setString(1, pass); s.setInt(2, userid);
int rows = s.executeUpdate();
```

2. **Dynamic types and implicit casting to and from strings:** PHP, like other scripting languages, has extensive support for string operations and automatic conversions between strings and other types. These features are handy for web applications because strings serve as the common medium between the browser, the web server, and the database backend. For example, we can convert a number into a string without an explicit cast:

```
if ($userid < 0) exit;
$query = "SELECT * from users
         WHERE userid = '$userid'";
```

3. **Variable scoping and the environment:** PHP has a number of mechanisms that minimize redundancy when accessing values from the execution environment. For example, HTTP *get* and *post* requests are automatically imported into the global name space as hash tables $_GET and $_POST. To access the "name" field of a submitted form, one can simply use $_GET['name'] directly in the program.

   If this still sounds like too much typing, PHP provides an extract operation that automatically imports all key-value pairs of a hash table into the current scope. In the example above, one can

[2]Both vulnerabilities have been reported to and fix ed by the PHP-fusion developers.

use extract(_GET, EXTR_OVERWRITE) to import data submitted using the HTTP `get` method. To access the $name field, one now simply types $name, which is preferred by some to $_GET['name'].

However, these conveniences come with security implications:

1. **SQL injection made easy:** Bind variables in Java have the benefit of assuring the programmer that any data passed into an SQL query remains data. The same cannot be said for the PHP example where malformed data from a malicious attacker may change the meaning of an SQL statement and cause unintended operations to the database. These are commonly called *SQL injection* attacks.

   In the example above (case 1), suppose $userid is controlled by the attacker and has value

   ```
   ' OR '1' = '1
   ```

   The query string becomes

   ```
   UPDATE users SET pass='...'
   WHERE userid='' OR '1'='1'
   ```

   which has the effect of updating the password for all users in the database.

2. **Unexpected conversions:** Consider the following code:

   ```
   if ($userid == 0) echo $userid;
   ```

   One would expect that if the program prints anything, it should be "0". Unfortunately, PHP implicitly casts string values into numbers before comparing them with an integer. Non-numerical values (e.g., "abc") convert to 0 without complaint, so the code above can print anything other than a non-zero number. We can imagine a potential SQL injection vulnerability if $userid is subsequently used to construct an SQL query as in the previous case.

3. **Uninitialized variables under user control:** In PHP, uninitialized variables default to null. Some programs rely on this fact for correct behavior; consider the following code:

   ```
   1  extract($_GET, EXTR_OVERWRITE);
   2  for ($i=0;$i<=7;$i++)
   3    $new_pass .= chr(rand(97, 122)); // append one char
   4  mysql_query("UPDATE ... $new_pass ...");
   ```

   This program generates a random password and inserts it into the database. However, due to the extract operation on line 1, a malicious user can introduce an arbitrary initial value for $new_pass by adding an unexpected new_pass field into the submitted HTTP form data.

```
CFG := build_control_flow_graph(AST);
foreach (basic_block b in CFG)
  summaries[b] := simulate_block(b);
return make_function_summary(CFG, summaries);
```

Figure 1: Pseudo-code for the analysis of a function.

3 Analysis

Given a PHP source file, our tool carries out static analysis in the following steps:

- We parse the PHP source into abstract syntax trees (ASTs). Our parser is based on the standard open-source implementation of PHP 5.0.5 [13]. Each PHP source file contains a *main* section (referred to as the *main* function hereafter although it is not part of any function definition) and zero or more user-defined functions. We store the user-defined functions in the environment and start the analysis from the main function.

- The analysis of a single function is summarized in Figure 1. For each function in the program, the analysis performs a standard conversion from the abstract syntax tree (AST) of the function body into a control flow graph (CFG). The nodes of the CFG are *basic blocks*: maximal single entry, single exit sequences of statements. The edges of the CFG are the jump relationships between blocks. For conditional jumps, the corresponding CFG edge is labeled with the branch predicate.

- Each basic block is simulated using symbolic execution. The goal is to understand the collective effects of statements in a block on the global state of the program and summarize their effects into a concise *block summary* (which describes, among other things, the set of variables that must be sanitized[3] before entering the block). We describe the simulation algorithm in Section 3.1.

- After computing a summary for each basic block, we use a standard reachability analysis to combine block summaries into a *function summary*. The function summary describes the pre- and post-conditions of a function (e.g., the set of sanitized input variables after calling the current function). We discuss this step in Section 3.2.

- During the analysis of a function, we might encounter calls to other user-defined functions. We discuss modeling function calls, and the order in which functions are analyzed, in Section 3.3.

[3]Sanitization is an operation that ensures that user input can be safely used in an SQL query (e.g., no unescaped quotes or spaces).

```
function simulate_block(BasicBlock b) : BlockSummary
{
  state := init_simulation_state();
  foreach (Statement s in b) {
    state := simulate(s, state);
    if (state.has_returned || state.has_exited)
      break;
  }
  summary := make_block_summary(state);
  return summary;
}
```

Figure 2: Pseudo-code for intra-block simulation.

$$
\begin{array}{rl}
\text{Type } (\tau) ::= & \mathsf{str} \mid \mathsf{bool} \mid \mathsf{int} \mid \top \\
\text{Const } (c) ::= & \mathsf{string} \mid \mathsf{int} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{null} \\
\text{L-val } (lv) ::= & x \mid \texttt{Arg\#i} \mid lv[e] \\
\text{Expr } (e) ::= & c \mid lv \mid e \;\mathsf{binop}\; e \mid \mathsf{unop}\; e \mid (\tau)e \\
\text{Stmt } (S) ::= & lv \leftarrow e \mid lv \leftarrow f(e_1, \ldots, e_n) \\
& \mid \mathsf{return}\; e \mid \mathsf{exit} \mid \mathsf{include}\; e
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{binop} & \in & \{+, -, \mathsf{concat}, ==, !=, <, >, \ldots\} \\
\mathsf{unop} & \in & \{-, \neg\}
\end{array}
$$

Figure 3: Language Definition

3.1 Simulating Basic Blocks

3.1.1 Outline

Figure 2 gives pseudo-code outlining the symbolic simulation process. Recall each basic block contains a linear sequence of statements with no jumps or jump targets in the middle. The simulation starts in an *initial state*, which maps each variable x to a symbolic initial value x_0. It processes each statement in the block in order, updating the simulator state to reflect the effect of that statement. The simulation continues until it encounters any of the following:

1. the end of the block;
2. a return statement. In this case, the current block is marked as a *return* block, and the simulator evaluates and records the return value;
3. an exit statement. In this case the current block is marked as an *exit* block;
4. a call to a user-defined function that exits the program. This condition is automatically determined using the function summary of the callee (see Sections 3.2 and 3.3).

Note that in the last case execution of the program has also terminated and therefore we remove any ensuing statements and outgoing CFG edges from the current block.

After a basic block is simulated, we use information contained in the final state of the simulator to summarize the effect of the block into a *block summary*, which we store for use during the intraprocedural analysis (see Section 3.2). The state itself is discarded after simulation.

The following subsections describe the simulation process in detail. We start with a definition of the subset of PHP that we model (§3.1.2) and discuss the representation of the simulation state and program values (§3.1.3, §3.1.4) during symbolic execution. Using the value representation, we describe how the analyzer simulates expressions (§3.1.5) and statements (§3.1.6). Finally, we describe how we represent and infer block summaries (§3.1.7).

3.1.2 Language

Figure 3 gives the definition of a small imperative language that captures a subset of PHP constructs that we believe is relevant to SQL injection vulnerabilities. Like PHP, the language is dynamically typed. We model three basic types of PHP values: strings, booleans and integers. In addition, we introduce a special $\top$ type to describe objects whose static types are undetermined (e.g., input parameters).[4]

Expressions can be *constants*, *l-values*, *unary* and *binary operations*, and *type casts*. The definition of l-values is worth mentioning because in addition to variables and function parameters, we include a named subscript operation to give limited support to the array and hash table accesses used extensively in PHP programs.

A statement can be an *assignment*, *function call*, *return*, *exit*, or *include*. The first four statement types require no further explanation. The include statement is a commonly used feature unique to scripting languages, which allows programmers to dynamically insert code into the program. In our language, include evaluates its string argument, and executes the program file designated by the string as if it is inserted at that program point (e.g., it shares the same scope). We describe how we simulate such behavior in Section 3.1.6.

3.1.3 State

Figure 4(a) gives the definition of values and states during simulation. The simulation state maps memory locations to their value representations, where a memory location is either a program variable (e.g. x), or an entry in a hash table accessed via another location (e.g. $x[key]$). Note the definition of locations is recursive, so multi-level hash dereferences are supported in our algorithm.

[4]In general, in a dynamically typed language, a more precise static approximation in this case would be a sum (aka. soft typing) [1, 20]. We have not found it necessary to use type sums in this work.

$\boxed{\text{Value Representation}}$

$$
\begin{aligned}
\text{Loc } (l) &::= x \mid l[\text{string}] \mid l[\top] \\
\text{Init-Values } (o) &::= l_0 \\
\text{Segment } (\beta) &::= \text{string} \mid \text{contains}(\sigma) \mid o \mid \bot \\
\text{String } (s) &::= \langle \beta_1, \ldots, \beta_n \rangle \\
\text{Boolean } (b) &::= \text{true} \mid \text{false} \mid \text{untaint}(\sigma_0, \sigma_1) \\
\text{Loc-set}(\sigma) &::= \{l_1, \ldots, l_n\} \\
\text{Integer } (i) &::= k \\
\text{Value } (v) &::= s \mid b \mid i \mid o \mid \top
\end{aligned}
$$

$\boxed{\text{Simulation State}}$

State (Γ) : Loc $\rightarrow$ Value

(a) Value representation and simulation state.

$\boxed{\text{Locations}}$

$$\frac{}{\Gamma \vdash x \overset{\text{Lv}}{\Rightarrow} x}\ \text{var} \qquad \frac{}{\Gamma \vdash \texttt{Arg\#n} \overset{\text{Lv}}{\Rightarrow} \texttt{Arg\#n}}\ \text{arg}$$

$$\frac{\Gamma \vdash e \overset{\text{E}}{\Rightarrow} l \qquad \Gamma \vdash e' \overset{\text{E}}{\Rightarrow} v' \quad v'' = \text{cast}(v', \text{str})}{\Gamma \vdash e[e'] \overset{\text{Lv}}{\Rightarrow} \begin{cases} l[\alpha] & \text{if } v'' = \langle \text{“}\alpha\text{”} \rangle \\ l[\top] & \text{otherwise} \end{cases}}\ \text{dim}$$

(b) L-values.

$\boxed{\text{Expressions}}$

Type casts:

$$\text{cast}(k, \text{bool}) = \begin{cases} \text{true} & \text{if } k \neq 0 \\ \text{false} & \text{otherwise} \end{cases}$$

$$\text{cast}(\text{true}, \text{str}) = \langle \text{“1”} \rangle$$

$$\text{cast}(\text{false}, \text{str}) = \langle \rangle$$

$$\text{cast}(v = \langle \beta_1, \ldots, \beta_n \rangle, \text{bool}) = \begin{cases} \text{true} & \text{if } (v \neq \langle \text{“0”} \rangle) \wedge \bigvee_{i=1}^{n} \neg\text{is\_empty}(\beta_i) \\ \text{false} & \text{if } (v = \langle \text{“0”} \rangle) \vee \bigwedge_{i=1}^{n} \text{is\_empty}(\beta_i) \\ \top & \text{otherwise} \end{cases}$$

...

Evaluation Rules:

$$\frac{\Gamma \vdash lv \overset{\text{Lv}}{\Rightarrow} l}{\Gamma \vdash lv \overset{\text{E}}{\Rightarrow} \Gamma(l)}\ \text{L-val}$$

$$\frac{\Gamma \vdash e_1 \overset{\text{E}}{\Rightarrow} v_1 \quad \text{cast}(v_1, \text{str}) = \langle \beta_1, \ldots, \beta_n \rangle \qquad \Gamma \vdash e_2 \overset{\text{E}}{\Rightarrow} v_2 \quad \text{cast}(v_2, \text{str}) = \langle \beta_{n+1}, \ldots, \beta_m \rangle}{\Gamma \vdash e_1 \text{ concat } e_2 \overset{\text{E}}{\Rightarrow} \langle \beta_1, \ldots, \beta_m \rangle}\ \text{concat}$$

$$\frac{\Gamma \vdash e \overset{\text{E}}{\Rightarrow} v \quad \text{cast}(v, \text{bool}) = v'}{\Gamma \vdash \neg e \overset{\text{E}}{\Rightarrow} \begin{cases} \text{true} & \text{if } v' = \text{false} \\ \text{false} & \text{if } v' = \text{true} \\ \text{untaint}(\sigma_1, \sigma_0) & \text{if } v' = \text{untaint}(\sigma_0, \sigma_1) \\ \top & \text{otherwise} \end{cases}}\ \text{not}$$

(c) Expressions.

Figure 4: Intrablock simulation algorithm.

On entry to the function, each location l is implicitly initialized to a symbolic initial value l_0, which makes up the initial state of the simulation. The values we represent in the state can be classified into three categories based on type:

Strings: Strings are the most fundamental type in many scripting languages, and precision in modeling strings directly determines analysis precision. Strings are typically constructed through concatenation. For example, user inputs (via HTTP `get` and `post` methods) are often concatenated with a pre-constructed skeleton to form an SQL query. Similarly, results from the query can be concatenated with HTML templates to form output. Modeling concatenation well enables an analysis to better understand information flow in a script. Thus, our string representations is based on concatenation. String values are represented as an ordered concatenation of string *segments*, which can be one of the following: a string constant, the initial value of a memory location on entry to the current block (l_0), or a string that contains initial values of zero or more elements from a set of memory locations (contains(σ)). We use the last representation to model return values from function calls, which may non-deterministically contain a combination of global variables and input parameters. For example, in

```
1  function f($a, $b) {
2      if (...) return $a;
3      else return $b;
4  }
5  $ret = f($x.$y, $z);
```

we represent the return value on line 5 as contains$(\{\text{x}, \text{y}, \text{z}\})$ to model the fact that it may contain any element in the set as a sub-string.

The string representation described above has the following benefits:

First, we get automatic constant folding for strings within the current block, which is often useful for resolving hash keys and distinguishing between hash references (e.g., in $key = “key”; return $hash[$key];).

Second, we can track how the contents of one input variable flow into another by finding occurrences of initial values of the former in the final representation of the latter. For example, in: $a = $a . $b, the final representation of $a is $\langle a_0, b_0 \rangle$. We know that if either $a or $b contains unsanitized user input on entry to the current block, so does $a upon exit.

Finally, interprocedural dataflow is possible by tracking function return values based on function summaries using contains(σ). We describe this aspect in more detail in Section 3.3.

Booleans: In PHP, a common way to perform input validation is to call a function that returns true or false depending on whether the input is well-formed or not. For example, the following code sanitizes $userid:

```
$ok = is_safe($userid);
if (!$ok) exit;
```

The value of Boolean variable $ok after the call is undetermined, but it is correlated with the safety of $userid. This motivates $\mathsf{untaint}(\sigma_0, \sigma_1)$ as a representation for such Booleans: σ_0 (resp. σ_1) represents the set of validated l-values when the Boolean is false (resp. true). In the example above, $ok has representation $\mathsf{untaint}(\{\}, \{\mathsf{userid}\})$.

Besides untaint, representation for Booleans also include constants (true and false) and unknown ($\top$).

Integers: Integer operations are less emphasized in our simulation. We track integer constants and binary and unary operations between them. We also support type casts from integers to Boolean and string values.

3.1.4 Locations and L-values

In the language definition in Figure 3, hash references may be aliased through assignments and l-values may contain hash accesses with non-constant keys. The same l-value may refer to different memory locations depending on the value of both the host and the key, and therefore, l-values are not suitable as memory locations in the simulation state.

Figure 4(b) gives the rules we use to resolve l-values into memory locations. The var and arg rules map each program variable and function argument to a memory location identified by its name, and the dim rule resolves hash accesses by first evaluating the hash table to a location and then appending the key to form the location for the hash entry.

These rules are designed to work in the presence of simple aliases. Consider the following program:

```
1 $hash = $_POST;
2 $key = 'userid';
3 $userid = $hash[$key];
```

The program first creates an alias ($hash) to hash table $_POST and then accesses the userid entry using that alias. On entry to the block, the initial state maps every location to its initial value:

$$\Gamma = \{\mathsf{hash} \Rightarrow \mathsf{hash}_0, \mathsf{key} \Rightarrow \mathsf{key}_0, \mathsf{\_POST} \Rightarrow \mathsf{\_POST}_0, \mathsf{\_POST[userid]} \Rightarrow \mathsf{\_POST[userid]}_0\}$$

According to the var rule, each variable maps to its own unique location. After the first two assignments, the state is:

$$\Gamma = \{\mathsf{hash} \Rightarrow \mathsf{\_POST}_0, \mathsf{key} \Rightarrow \langle\text{‘userid’}\rangle, \ldots\}$$

We use the dim rule to resolve $hash[$key] on line 3: $hash evaluates to $\mathsf{\_POST}_0$, and $key evaluates to constant string 'userid'. Therefore, the l-value $hash[$key] evaluates to location _POST[userid], and thus the analysis assigns the desired value $\mathsf{\_POST[userid]}_0$ to $userid.

3.1.5 Expressions

We perform abstract evaluation of expressions based on the value representation described above. Because PHP is a dynamically typed language, operands are implicitly cast to appropriate types for operations in an expression. Figure 4(c) gives a representative sample of cast rules simulating cast operations in PHP. For example, Boolean value true, when used in a string context, evaluates to “1”. false, on the other hand, is converted to the empty string instead of “0”. In cases where exact representation is not possible, the result of the cast is unknown ($\top$).

Figure 4(c) also gives three representative rules for evaluating expressions. The first rule handles l-values, and the result is obtained by first resolving the l-value into a memory location, and then looking up the location in the evaluation context (recall that $\Gamma(l) = l_0$ on entry to the block).

The second rule models string concatenation. We first cast the value of both operands to string values, and the result is the concatenation of both.

The final rule handles Boolean negation. The interesting case involves untaint values. Recall that $\mathsf{untaint}(\sigma_0, \sigma_1)$ denotes an unknown Boolean value that is false (resp. true) if l-values in the set σ_0 (resp. σ_1) are sanitized. Given this definition, the negation of $\mathsf{untaint}(\sigma_0, \sigma_1)$ is $\mathsf{untaint}(\sigma_1, \sigma_0)$.

The analysis of an expression is $\top$ if we cannot determine a more precise representation, which is a potential source of false negatives.

3.1.6 Statements

We model assignments, function calls, return, exit, and include statements in the program. The assignment rule resolves the left-hand side to a memory location l, and evaluates the right-hand side to a value v. The updated simulation state after the assignment maps l to the new value v:

$$\frac{\Gamma \vdash lv \overset{\mathtt{Lv}}{\Rightarrow} l \qquad \Gamma \vdash e \overset{\mathtt{E}}{\Rightarrow} v}{\Gamma \vdash lv \leftarrow e \overset{\mathtt{S}}{\Rightarrow} \Gamma[l \mapsto v]} \ \mathsf{assignment}$$

Function calls are similar. The return value of a function call $f(e_1, \ldots, e_n)$ is modeled using either $\mathsf{contains}(\sigma)$ (if f returns a string) or $\mathsf{untaint}(\sigma_0, \sigma_1)$ (if f returns a Boolean) depending on the inferred summary for f. We defer discussion of the function summaries and the return value representation to Sections 3.2 and 3.3. For the purpose of this section, we use the uninterpreted value $f(v_1, \ldots, v_n)$ as a place holder for the actual representation of the return value:

$$\frac{\Gamma \vdash lv \overset{\mathtt{Lv}}{\Rightarrow} l \quad \Gamma \vdash e_1 \overset{\mathtt{E}}{\Rightarrow} v_1 \ \ldots \ \Gamma \vdash e_n \overset{\mathtt{E}}{\Rightarrow} v_n}{\Gamma \vdash lv \leftarrow f(e_1, \ldots, e_n) \overset{\mathtt{S}}{\Rightarrow} \Gamma[l \mapsto f(v_1, \ldots, v_n)]} \ \mathsf{fun}$$

In addition to the return value, certain functions have pre- and post-conditions depending on the operation they perform. Pre- and post-conditions are inferred and stored in the callee's summary, which we describe in detail in Sections 3.2 and 3.3. Here we show two examples to illustrate their effects:

```
1  function validate($x) {
2   if (!is_numeric($x)) exit;
3   return;
4  }
5  function my_query($q) {
6    global $db;
7    mysql_db_query($db, $q);
8  }
9  validate($a.$b);
10 my_query("SELECT...WHERE a = '$a' AND c = '$c'");
```

The validate function tests whether the argument is a number (and thus safe) and aborts if it is not. Therefore, line 9 sanitizes both $a and $b. We record this fact by inspecting the value representation of the actual parameter (in this case $\langle a_0, b_0 \rangle$), and remembering the set of non-constant segments that are sanitized.

The second function my_query uses its argument as a database query string by calling mysql_db_query. To prevent SQL injection attacks, any user input must be sanitized before it becomes part of the first parameter. Again, we enforce this requirement by inspecting the value representation of the actual parameter. We record any unsanitized non-constant segments (in this case $c, since $a is sanitized on line 9) and require they be sanitized as part of the pre-condition for the current block.

Sequences of assignments and function calls are simulated by using the output environment of the previous statement as the input environment of the current statement:

$$\frac{\Gamma \vdash s_1 \overset{\mathsf{S}}{\Rightarrow} \Gamma' \quad \Gamma' \vdash s_2 \overset{\mathsf{S}}{\Rightarrow} \Gamma''}{\Gamma \vdash (s_1; s_2) \overset{\mathsf{S}}{\Rightarrow} \Gamma''} \ \text{seq}$$

The final simulation state is the output state of the final statement.

The return and exit statements terminate control flow[5] and require special treatment. For a return, we evaluate the return value and use it in calculating the function summary. In case of an exit statement, we mark the current block as an *exit block*.

Finally, include statements are a commonly used feature unique to scripting languages allowing programmers to dynamically insert code and function definitions from another script. In PHP, the included code inherits the variable scope at the point of the include statement. It may introduce new variables and function definitions, and change or sanitize existing variables before the next statement in the block is executed.

We process include statements by first parsing the included file, and adding any new function definitions to the environment. We then splice the control flow graph of the included main function at the current program point by a) removing the include statement, b) breaking the current basic block into two at that point, c) linking the first half of the current block to the start of the main function, and all return blocks (those containing a return statement) in the included CFG to the second half, and d) replacing the return statements in the included script with assignments to reflect the fact that control flow resumes in the current script.

[5]So do function calls that exits the program, in which case we remove any ensuing statements and outgoing edges from the current CFG block. See Section 3.3.

3.1.7 Block summary

The final step for the symbolic simulator is to characterize the behavior of a CFG block into a concise summary. A block summary is represented as a six-tuple $\langle \mathcal{E}, \mathcal{D}, \mathcal{F}, \mathcal{T}, \mathcal{R}, \mathcal{U} \rangle$:

- **Error set ($\mathcal{E}$):** the set of input variables that must be sanitized before entering the current block. These are accumulated during simulation of function calls that require sanitized input.
- **Definitions ($\mathcal{D}$):** the set of memory locations defined in the current block. For example, in

  ```
  $a = $a.$b; $c = 123;
  ```

  we have $\mathcal{D} = \{a, c\}$.
- **Value flow ($\mathcal{F}$):** the set of pairs of locations (l_1, l_2) where the string value of l_1 on entry becomes a substring of l_2 on exit. In the example above, $\mathcal{F} = \{(a, a), (b, a)\}$.
- **Termination predicate ($\mathcal{T}$):** true if the current block contains an exit statement, or if it calls a function that causes the program to terminate.
- **Return value ($\mathcal{R}$):** records the representation for the return value if any, undefined otherwise. Note that if the current block has no successors, either $\mathcal{R}$ has a value or $\mathcal{T}$ is true.
- **Untaint set ($\mathcal{U}$):** for each successor of the current CFG block, we compute the set of locations that are sanitized if execution continues onto that block. Sanitization can occur via function calls, casting to safe types (e.g., int, etc), regular expression matching, and other tests. The untaint set for different successors might differ depending on the value of branch predicates. We show an example below.

  ```
  validate($a);
  $b = (int) $c;
  if (is_numeric($d))
        ...
  ```

  As mentioned earlier, validate exits if $a is unsafe. Casting to integer also returns a safe result. Therefore, the untaint set is $\{a, b, d\}$ for the true branch, and $\{a, b\}$ for the false branch.

3.2 Intraprocedural Analysis

Based on block summaries computed in the previous step, the intraprocedural analysis computes the following summary $\langle \mathcal{E}, \mathcal{R}, \mathcal{S}, \mathcal{X} \rangle$ for each function:

1. **Error set ($\mathcal{E}$):** the set of memory locations (variables, parameters, and hash accesses) whose value may flow into a database query, and therefore must be sanitized before invoking the current function. For the main function, the error set must not include any user-defined variables (e.g. $_GET['...'] or $_POST['...'])—the analysis emits an error message for each such violation.

   We compute $\mathcal{E}$ by a backwards reachability analysis that propagates the error set of each block (using the $\mathcal{E}, \mathcal{D}, \mathcal{F}$, and $\mathcal{U}$ components in the block summaries) to the start block of the function.

2. **Return set ($\mathcal{R}$):** the set of parameters or global variables whose value may be a substring of the return value of the function. $\mathcal{R}$ is only computed for functions that may return string values. For example, in the following code, the return set includes both function arguments and the global variable $table (i.e., $\mathcal{R} = \{\text{table}, \texttt{Arg\#1}, \texttt{Arg\#2}\}$).

```
function make_query($user, $pass) {
  global $table;
  return "SELECT * from $table ".
    "where user = $user and pass = $pass";
}
```

   We compute the function return set by using a forward reachability analysis that expresses each return value (recorded in the block summaries as $\mathcal{R}$) as a set of function parameters and global variables.

3. **Sanitized values ($\mathcal{S}$):** the set of parameters or global variables that are sanitized on function exit. We compute the set by using a forward reachability analysis to determine the set of sanitized inputs at each return block, and we take the intersection of those sets to arrive at the final result.

   If the current function returns a Boolean value as its result, we distinguish the sanitized value set when the result is true versus when it is false (mirroring the untaint representation for Boolean values above). The following example motivates this distinction:

```
function is_valid($x) {
  if (is_numeric($x)) return true;
  return false;
}
```

   The parameter is sanitized if the function returns true, and the return value is likely to be used by the caller to determine the validity of user input. In the example above,

$$\mathcal{S} = (\text{false} \Rightarrow \{\}, \text{true} \Rightarrow \{\texttt{Arg\#1}\})$$

   For comparison, the validate function defined previously has $\mathcal{S} = (* \Rightarrow \{\texttt{Arg\#1}\})$. In the next section, we describe how we make use of this information in the caller.

4. **Program Exit ($\mathcal{X}$):** a Boolean which indicates whether the current function terminates program execution on all paths. Note that control flow can leave a function either by returning to the caller or by terminating the program. We compute the exit predicate by enumerating over all CFG blocks that have no successors, and identify them as either return blocks or exit blocks (the $\mathcal{T}$ and $\mathcal{R}$ component in the block summary). If there are no return blocks in the CFG, the current function is an exit function.

The dataflow algorithms used in deriving these facts are fairly standard fix-point computations. We omit the details for brevity.

3.3 Interprocedural Analysis

This section describes how we conduct interprocedural analysis using summaries computed in the previous step. Assuming f has summary $\langle \mathcal{E}, \mathcal{R}, \mathcal{S}, \mathcal{X} \rangle$, we process a function call $f(e_1, \ldots, e_n)$ during intrablock simulation as follows:

1. **Pre-conditions:** We use the error set ($\mathcal{E}$) in the function summary to identify the set of parameters and global variables that must be sanitized before calling this function. We substitute actual parameters for formal parameters in $\mathcal{E}$ and record any unsanitized non-constant segments of strings in the error set as the sanitization pre-condition for the current block.

2. **Exit condition:** If the callee is marked as an exit function (i.e., $\mathcal{X}$ is true), we remove any statements that follow the call and delete all outgoing edges from the current block. We further mark the current block as an exit block.

3. **Post-conditions:** If the function unconditionally sanitizes a set of input parameters and global variables, we mark this set of values as safe in the simulation state after substituting actual parameters for formal parameters.

   If sanitization is conditional on the return value (e.g., the is_valid function defined above), we record the intersection of its two component sets as being unconditionally sanitized (i.e., $\sigma_0 \cap \sigma_1$ if the untaint set is $(\mathsf{false} \Rightarrow \sigma_0, \mathsf{true} \Rightarrow \sigma_1)$).

4. **Return value:** If the function returns a Boolean value and it conditionally sanitizes a set of input parameters and global variables, we use the untaint representation to model that correlation:

$$\frac{\begin{array}{c}\Gamma \vdash lv \overset{\texttt{Lv}}{\Rightarrow} l \quad \Gamma \vdash e_1 \overset{\texttt{E}}{\Rightarrow} v_1 \ \ldots \ \Gamma \vdash e_n \overset{\texttt{E}}{\Rightarrow} v_n \\ \texttt{Summary}(f) = \langle \mathcal{E}, \mathcal{R}, \mathcal{S}, \mathcal{X} \rangle \\ \mathcal{S} = (\mathsf{false} \Rightarrow \sigma_0, \mathsf{true} \Rightarrow \sigma_1) \quad \sigma_* = \sigma_0 \cap \sigma_1 \\ \sigma_0' = \mathsf{subst}_{\bar{v}}(\sigma_0 - \sigma_*) \quad \sigma_1' = \mathsf{subst}_{\bar{v}}(\sigma_1 - \sigma_*)\end{array}}{\Gamma \vdash lv \leftarrow f(e_1, \ldots, e_n) \overset{\texttt{S}}{\Rightarrow} \Gamma[l \mapsto \mathsf{untaint}(\sigma_0', \sigma_1')]}$$

In the rule above, $\mathsf{subst}_{\bar{v}}(\sigma)$ substitutes actual parameters (v_i) for formal parameters in σ.

If the callee returns a string value, we use the return set component of the function summary ($\mathcal{R}$) to determine the set of input parameters and global variables that might become a substring of the return value:

$$\frac{\begin{array}{c}\Gamma \vdash lv \overset{\texttt{Lv}}{\Rightarrow} l \quad \Gamma \vdash e_1 \overset{\texttt{E}}{\Rightarrow} v_1 \ \ldots \ \Gamma \vdash e_n \overset{\texttt{E}}{\Rightarrow} v_n \\ \texttt{Summary}(f) = \langle \mathcal{E}, \mathcal{R}, \mathcal{S}, \mathcal{X} \rangle \quad \sigma' = \mathsf{subst}_{\bar{v}}(\mathcal{R})\end{array}}{\Gamma \vdash lv \leftarrow f(e_1, \ldots, e_n) \overset{\texttt{S}}{\Rightarrow} \Gamma[l \mapsto \mathsf{contains}(\sigma')]}$$

Since we require the summary information of a function before we can analyze its callers, the order in which functions are analyzed is important. Due to the dynamic nature of PHP (e.g., include statements), we analyze functions on demand—a function *f* is analyzed and summarized when we first encounter a call to *f*. The summary is then memoized to avoid redundant analysis. Effectively, our algorithm analyzes the source codebase in topological order based on the static function call graph. If we encounter a cycle during the analysis, the current implementation uses a dummy "no-op" summary as a model for the second invocation (i.e., we do not compute fix points for recursive functions). In theory, this is a potential source of false negatives, which can be removed by adding a simple iterative algorithm that handles recursion. However, practically, such an algorithm may be unnecessary given the rare occurrence of recursive calls in PHP programs.

4 Experimental Results

The analysis described in Section 3 has been implemented as two separate parts: a frontend based on the open source PHP 5.0.5 distribution that parses the source files into abstract syntax trees and a backend written in OCaml [8] that reads the ASTs into memory and carries out the analysis. This separation ensures maximum compatibility while minimizing dependence on the PHP implementation.

The decision to use different levels of abstraction in the intrablock, intraprocedural, and interprocedural levels enabled us to fine tune the amount of information we retain at one level independent of the algorithm used in another and allowed us to quickly build a usable tool. The checker is largely automatic and requires little human intervention for use. We seed the checker with a small set of query functions (e.g. mysql_query) and sanitization operations (e.g. is_numeric). The checker infers the rest automatically.

Regular expression matching presents a challenge to automation. Regular expressions are used for a variety of purposes including, but not limited to, input validation. Some regular expressions match well-formed input while others detect malformed input; assuming one way or the other results in either false positives or false negatives. Our solution is to maintain a database of previously seen regular expressions and their effects, if any. Previously unseen regular expressions are assumed by default to have no sanitization effects, so as not to miss any errors due to incorrect judgment. To make it easy for the user to specify the sanitization effects of regular expressions, the checker has an interactive mode where the user is prompted when the analysis encounters a previously unseen regular expression and the user's answers are recorded for future reference.[6] Having the user declare the role of regular expressions has the real potential to introduce errors into the analysis; however, practically, we found this approach to be very effective and it helped us find at least two vulnerabilities caused by overly lenient regular expressions being used for sanitization.[7] Our tool collected information for 49 regular expressions from the user over all our experiments (the user replies with one keystroke for each inquiry), so the burden on the user is minimal.

The checker detects errors by using information from the summary of the main function—the checker marks all variables that are required to be sanitized on entry as potential security vulnerabilities. From the checker's perspective, these variables are defined in the environment and used to construct SQL queries without being sanitized. In reality, however, these variables are either defined by the runtime environment or by some language constructs that the checker does not fully understand (e.g., the extract operation in PHP which we describe in a case study below). The tool emits an *error* mes-

[6]Here we assume that a regular expression used to sanitize input in one context will have the same effect in another, which, based on our experience, is the common case. Our implementation now provides paranoid users with a special switch that ignores recorded answers and repeatedly ask the user the same question over and over if so desired.

[7]For example, Utopia News Pro misused "`[0-9]+`" to validate some user input. This regular expression only checks that the string contains a number, instead of ensuring that the input *is* actually a number. The correct regular expression in this case is "`^[0-9]+$`".

Application (KLOC)	**Err Msgs**	**Bugs (FP)**	**Warn**
News Pro (6.5)	8	8 (0)	8
myBloggie (9.2)	16	16 (0)	23
PHP Webthings (38.3)	20	20 (0)	6
DCP Portal (121)	39	39 (0)	55
e107 (126)	16	16 (0)	23
Total	99	99 (0)	115

Table 1: Summary of experiments. LOC statistics include embedded HTML, and thus is a rough estimate of code complexity. Err Msgs: number of reported errors. Bugs: number of confirmed bugs from error reports. FP: number of false positives. Warn: number of unique warning messages for variables of unresolved origin (uninspected).

sage if the variable is known to be controlled by the user (e.g. `$_GET['...']`, `$_POST['...']`, `$_COOKIE['...']`, etc). For others, the checker emits a *warning*.

We conducted our experiments on the latest versions of six open source PHP code bases: `e107 0.7`, `Utopia News Pro 1.1.4`, `mybloggie 2.1.3beta`, `DCP Portal v6.1.1`, `PHP Webthings 1.4patched`, and `PHP fusion 6.00.204`. Table 1 summarizes our findings for the first five. The analysis terminates within seconds for each script examined (which may dynamically include other source files). Our checker emitted a total of 99 error messages for the first five applications, where unsanitized user input (from `$_GET`, `$_POST`, etc) may flow into SQL queries. We manually inspected the error reports and believe all 99 represent real vulnerabilities.[8] We have notified the developers about these errors and will publish security advisories once the errors have been fixed. We have not inspected warning messages—unsanitized variables of unresolved origin (e.g. from database queries, configuration files, etc) that are subsequently used in SQL queries due to the high likelihood of false positives.

PHP-fusion is different from the other five code bases because it does not directly access HTTP form data from input hash tables such as `$_GET` and `$_POST`. Instead it uses the `extract` operation to automatically import such information into the current variable scope. We describe our findings for PHP-fusion in the following subsection.

4.1 Case Study: Two Exploitable SQL Injection Attacks in PHP-fusion

In this section, we show two case studies of exploitable SQL injection vulnerabilities in PHP-fusion detected by our tool. PHP-fusion is an open-source content management system (CMS) built on PHP and MySQL. Excluding locale specific customization modules, it consists of over 16,000 lines of PHP code and has a wide user-base because of its speed, customizability and rich features. Browsing through the code, it is obvious that the author programmed with security in mind and has taken extra care in sanitizing input before use in query strings.

Our experiments were conducted on the then latest 6.00.204 version of the software. Unlike other code bases we have examined, PHP-fusion uses the `extract` operation to import user input into the current scope. As an example, `extract($_POST, EXTR_OVERWRITE)` has the effect of introducing one variable for each key in the `$_POST` hash table into the current scope, and assigning the value of `$_POST[key]` to that variable. This feature reduces typing, but introduces confusion for the checker and security vulnerabilities into the software—both of the exploits we constructed involve use of uninitialized variables whose values can be manipulated by the user because of the `extract` operation.

Since PHP-fusion does not directly read user input from input hashes such as `$_GET`or `$_POST`, there are no direct error messages generated by our tool. Instead we inspect warnings (recall the discussion about errors and warnings above), which correspond to security sensitive variables whose definition is unresolved by the checker (e.g., introduced via the `extract` operation, or read from configuration files).

We ran our checker on all top level scripts in PHP-fusion. The tool generated 22 unique warnings, a majority of which relate to configuration variables that are used in the construction of a large number of queries.[9] After filtering those out, 7 warnings in 4 different files remain.

We believe all but one of the 7 warnings may result in exploitable security vulnerabilities. The lone false positive arises from an unanticipated sanitization:

```
/* php-files/lostpassword.php */
if (!preg_match("/^[0-9a-z]{32}$/", $account))
        $error = 1;
if (!$error) { /* database access using $account */ }
if ($error) redirect("index.php");
```

Instead of terminating the program immediately based on the result from `preg_match`, the program sets the `$error` flag to `true` and delays error handling, which is in general not a good practice. This idiom can be handled by adding slightly more information in the block summary.

We investigated the first two of the remaining warnings for potential exploits and confirmed that both are indeed exploitable on a test installation. Unsurprisingly

[8]Information about the results, along with the source codebases, are available online at:
`http://glide.stanford.edu/yichen/research/`.

[9]Database configuration variables such as `$db_prefix` accounted for 3 false positives, and information derived from the database queries and configuration settings (e.g. locale settings) caused the remaining 12.

both errors are made possible because of the extract operation. We explain these two errors in detail below.

1) Vulnerability in script for recovering lost password. This is a remotely exploitable vulnerability that allows any registered user to elevate his privileges via a carefully constructed URL. We show the relevant code below:

```
1  /* php-files/lostpassword.php */
2  for ($i=0;$i<=7;$i++)
3      $new_pass .= chr(rand(97, 122));
4  ...
5  $result = dbquery("UPDATE ".$db_prefix."users
6    SET user_password=md5('$new_pass')
7    WHERE user_id='".$data['user_id']."'");
```

Our tool issued a warning for $new_pass, which is uninitialized on entry and thus defaults to the empty string during normal execution. The script proceeds to add seven randomly generated letters to $new_pass (lines 2-3), and uses that as the new password for the user (lines 5-7). The SQL request under normal execution takes the following form:

```
UPDATE users SET user_password=md5('???????')
  WHERE user_id='userid'
```

However, a malicious user can simply add a new_pass field to his HTTP request by appending, for example, the following string to the URL for the password reminder site:

```
&new_pass=abc%27%29%2cuser_level=%27103%27%2cuser_aim=%28%27
```

The extract operation described above will magically introduce $new_pass in the current variable scope with the following initial value:

```
abc'), user_level =' 103', user_aim = ('
```

The SQL request is now constructed as:

```
UPDATE users SET user_password=md5('abc'),
     user_level='103', user_aim=('???????')
  WHERE user_id='userid'
```

Here the password is set to "abc", and the user privilege is elevated to 103, which means "Super Administrator." The newly promoted user is now free to manipulate any content on the website.

2) Vulnerability in the messaging sub-system. This vulnerability exploits another use of potentially uninitialized variable $result_where_message_id in the messaging sub system. We show the relevant code in Figure 5.

Our tool warns about unsanitized use of $result_where_message_id. On normal input, the program initializes $result_where_message_id using a cascading `if` statement. As shown in the code, the author is very careful about sanitizing values that are used to construct $result_where_message_id. However, the cascading sequence of `if` statements does not have a default branch. And therefore, $result_where_message_id might be uninitialized on malformed input. We exploit this fact, and append

```
&request_where_message_id=1=1/*
```

The query string submitted on line 11-13 thus becomes:

```
1  if (isset($msg_view)) {
2    if (!isNum($msg_view)) fallback("messages.php");
3    $result_where_message_id="message_id=".$msg_view;
4  } elseif (isset($msg_reply)) {
5    if (!isNum($msg_reply)) fallback("messages.php");
6    $result_where_message_id="message_id=".$msg_reply;
7  }
8  ... /* ~100 lines later */ ...
9  } elseif (isset($_POST['btn_delete']) ||
10   isset($msg_delete)) { // delete message
11   $result = dbquery("DELETE FROM ".$db_prefix.
12    "messages WHERE ".$result_where_message_id. // BUG
13    " AND ".$result_where_message_to);
```

Figure 5: An exploitable vulnerability in PHP-fusion 6.00.204.

```
DELETE FROM messages WHERE 1=1 /* AND ...
```

Whatever follows "/*" is treated as comments in MySQL and thus ignored. The result is loss of all private messages in the system. Due to the complex control and data flow, this error is unlikely to be discovered via code review or testing.

We reported both exploits to the author of PHP-fusion, who immediately fixed these vulnerabilities and released a new version of the software.

5 Related Work

5.1 Static techniques

WebSSARI is a type-based analyzer for PHP [7]. It uses a simple intraprocedural tainting analysis to find cases where user controlled values flow into functions that require trusted input (i.e. *sensitive functions*). The analysis relies on three user written "prelude" files to provide information regarding: 1) the set of all sensitive functions–those require sanitized input; 2) the set of all untainting operations; and 3) the set of untrusted input variables. Incomplete specification results in both substantial numbers of false positives and false negatives.

WebSSARI has several key limitations that restrict the precision and analysis power of the tool:

1. WebSSARI uses an intraprocedural algorithm and thus only models information flow that does not cross function boundaries.

   Large PHP codebases typically define a number of application specific subroutines handling common operations (e.g., query string construction, authentication, sanitization, etc) using a small number of system library functions (e.g., mysql_query). Our algorithm is able to automatically infer information flow and pre- and post-conditions for such user-defined functions whereas WebSSARI relies on the user to specify the constraints of each, a significant burden that needs to be repeated for each source codebase examined. Examples in Section 3.3 represent some common forms of user-defined functions that WebSSARI is not able to model without annotations.

To show how much interprocedural analysis improves the accuracy of our analysis, we turned off function summaries and repeated our experiment on `News Pro`, the smallest of the five codebases. This time, the analysis generated 19 error messages (as opposed to 8 with interprocedural analysis). Upon inspection, all 11 extra reports are false positives due to user-defined sanitization operations.

2. WebSSARI does not seem to model conditional branches, which represent one of the most common forms of sanitization in the scripts we have analyzed. For example, we believe it will report a false warning on the following code:

```
if (!is_numeric($_GET['x']))
    exit;
mysql_query("... $_GET['x'] ...'');
```

Furthermore, interprocedural conditional sanitization (see the example in Section 3.1.6) is also fairly common in codebases.

3. WebSSARI uses an algorithm based on static types that does not specifically model dynamic features in scripts. For example, dynamic typing may introduce subtle errors that WebSSARI misses. The include statement, used extensively in PHP scripts, dynamically inserts code to the program which may contain, induce, or prevent errors.

We are unable to directly compare the experimental results due to the fact that neither the bug reports nor the WebSSARI tool are available publicly. Nor are we able to compare false positive rates since WebSSARI reports per-file statistics which may underestimate the false positive ratio. A file with 100 false positives and 1 real bug is considered to be "vulnerable" and therefore does not contribute to the false positive rate computed in [7].

Livshits and Lam [9] develop a static detector for security vulnerabilities (e.g., SQL injection, cross site scripting, etc) in Java applications. The algorithm uses a BDD-based context-sensitive pointer analysis [19] to find potential flow from untrusted sources (e.g., user input) to trusting sinks (e.g., SQL queries). One limitation of this analysis is that it does not model control flow in the program and therefore may misflag sanitized input that subsequently flows into SQL queries. Sanitization with conditional branching is common in PHP programs, so techniques that ignore control flow are likely to cause large numbers of false positives on such code bases.

Other tainting analysis that are proven effective on C code include CQual [4], MECA [21], and MC [6, 2]. Collectively they have found hundreds of previously unknown security errors in the Linux kernel.

Christensen *et. al.* [3] develop a string analysis that approximates string values in a Java program using a context free grammar. The result is widened into a regular language and checked against a specification of expected output to determine syntactic correctness. However, syntactic correctness does not entail safety, and therefore it is unclear how to adapt this work to the detection of SQL injection vulnerabilities. Minamide [10] extends the approach and construct a string analyzer for PHP, citing SQL injection detection as a possible application. However, the analyzer models a small set of string operations in PHP (e.g., concatenation, string matching and replacement) and ignores more complex features such as dynamic typing, casting, and predicates. Furthermore, the framework only seems to model sanitization with string replacement, which represents a small subset of all sanitization in real code. Therefore, accurately pinpointing injection attacks remains challenging.

Gould *et. al.* [5] combines string analysis with type checking to ensure not only syntactic correctness but also type correctness for SQL queries constructed by Java programs. However, type correctness does not imply safety, which is the focus of our analysis.

5.2 Dynamic Techniques

Scott and Sharp [15] propose an application-level firewall to centralize sanitization of client input. Firewall products are also commercially available from companies such as NetContinuum, Imperva, Watchfire, etc. Some of these firewalls detect and guard against previously known attack patterns, while others maintain a white list of valid inputs. The main limitation here is that the former is susceptible to both false positives and false negatives, and the latter is reliant on correct specifications, which are difficult to come by.

The Perl taint mode [12] enables a set of special security checks during execution in an unsafe environment. It prevents the use of untrusted data (e.g., all command line arguments, environment variables, data read from files, etc) in operations that require trusted input (e.g., any command that invokes a sub-shell). Nguyen-Tuong [11] proposes a taint mode for PHP, which, unlike the Perl taint mode, not define sanitizing operations. Instead, it tracks each character in the user input individually, and employs a set of heuristics to determine whether a query is safe when it contains fragments of user input. For example, among others, it detects an injection if an operator symbol (e.g., "(", ")", "%", etc) is marked as tainted. This approach is susceptible to both false positives and false negatives. Note that static analyses are also susceptible to both false positives and false negatives. The key distinction is that in static analyses, inaccuracies are resolved at compile time instead of at runtime, which is much less forgiving.

6 Conclusion

We have presented a static analysis algorithm for detecting security vulnerabilities in PHP. Our analysis employs a novel three-tier architecture that enables us to handle dynamic features unique to scripting languages such as dynamic typing and code inclusion. We demonstrate the effectiveness of our approach by running our tool on six popular open source PHP code bases and finding 105 previously unknown security vulnerabilities, most of which we believe are remotely exploitable.

Acknowledgement

This research is supported in part by NSF grants SA4899-10808PG, CCF-0430378, and an IBM Ph.D. fellowship. We would like to thank our shepherd Andrew Myers and the anonymous reviewers for their helpful comments and feedback.

References

[1] A. Aiken, E. Wimmers, and T. Lakshman. Soft typing with conditional types. In *Proceedings of the 21st Annual Symposium on Principles of Programming Languages*, 1994.

[2] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *2002 IEEE Symposium on Security and Privacy*, 2002.

[3] A. Christensen, A. Moller, and M. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th Static Analysis Symposium*, 2003.

[4] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, June 2002.

[5] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 26th International Conference on Software Engineering*, 2004.

[6] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[7] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International World Wide Web Conference*, 2004.

[8] X. Leroy, D. Doligez, J. Garrigue, and J. Vouillon. The Objective Caml system. Software and documentation available on the web, `http://caml.inria.fr`.

[9] V. Livshits and M. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, 2005.

[10] Y. Minamide. Approximation of dynamically generated web pages. In *Proceedings of the 14th International World Wide Web Conference*, 2005.

[11] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the 20th International Information Security Conference*, 2005.

[12] Perl documentation: Perlsec. `http://search.cpan.org/dist/perl/pod/perlsec.pod`.

[13] PHP: Hypertext Preprocessor. `http://www.php.net`.

[14] PHP usage statistics. `http://www.php.net/usage.php`.

[15] D. Scott and R. Sharp. Abstracting application-level web security. In *Proceedings of the 11th International World Wide Web Conference*, 2002.

[16] Security space apache module survey (Oct 2005). `http://www.securityspace.com/s_survey/data/man.200510/apachemods.html`.

[17] Symantec Internet security threat report: Vol. VII. Technical report, Symantec Inc., Mar. 2005.

[18] TIOBE programming community index for November 2005. `http://www.tiobe.com/tpci.htm`.

[19] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 2004.

[20] A. Wright and R. Cartwright. A practical soft type system for Scheme. *ACM Trans. Prog. Lang. Syst.*, 19(1):87–152, Jan. 1997.

[21] J. Yang, T. Kremenek, Y. Xie, and D. Engler. MECA: an extensible, expressive system and language for statically checking security properties. In *Proceedings of the 10th Conference on Computer and Communications Security*, 2003.

Rule-Based Static Analysis of Network Protocol Implementations

Octavian Udrea, Cristian Lumezanu, and Jeffrey S. Foster
{udrea, lume, jfoster}@cs.umd.edu
University of Maryland, College Park

Abstract

Today's software systems communicate over the Internet using standard protocols that have been heavily scrutinized, providing some assurance of resistance to malicious attacks and general robustness. However, the software that implements those protocols may still contain mistakes, and an incorrect implementation could lead to vulnerabilities even in the most well-understood protocol. The goal of this work is to close this gap by introducing a new technique for checking that a C implementation of a protocol matches its description in an RFC or similar standards document. We present a static (compile-time) source code analysis tool called Pistachio that checks C code against a rule-based specification of its behavior. Rules describe what should happen during each round of communication, and can be used to enforce constraints on ordering of operations and on data values. Our analysis is not guaranteed sound due to some heuristic approximations it makes, but has a low false negative rate in practice when compared to known bug reports. We have applied Pistachio to implementations of SSH and RCP, and our system was able to find many bugs, including security vulnerabilities, that we confirmed by hand and checked against each project's bug databases.

1 Introduction

Networked software systems communicate using protocols designed to provide security against attacks and robustness against network glitches. There has been a significant body of research, both formal and informal, in scrutinizing abstract protocols and proving that they meet certain reliability and safety requirements [24, 18, 6, 14, 25]. These abstract protocols, however, are ultimately implemented in software, and an incorrect implementation could lead to vulnerabilities even in the most heavily-studied and well-understood protocol.

In this paper we present a tool called Pistachio that helps close this gap. Pistachio is a static (compile-time) analysis tool that can check that each communication step taken by a protocol implementation matches an abstract specification. Because it starts from a detailed protocol specification, Pistachio is able to check communication properties that generic tools such as buffer overflow detectors do not look for. Our static analysis algorithm is also very fast, enabling Pistachio to be deployed regularly during the development cycle, potentially on every compile.

The input to our system is the C source code implementing the protocol and a *rule-based* specification of its behavior, where each rule describes what should happen in a "round" of communication. For example, the IETF current draft of the SSH connection protocol specifies that "When either party wishes to terminate the channel, it sends *SSH_MSG_CHANNEL_CLOSE*. Upon receiving this message, a party *must* send back a *SSH_MSG_CHANNEL_CLOSE*...." This statement translates into the following rule (slightly simplified):

$$\begin{array}{l}
\texttt{recv(\_, } \mathit{in}\texttt{, \_)} \\
\mathit{in}[0] = \mathit{SSH\_MSG\_CHANNEL\_CLOSE} \\
\Rightarrow \\
\texttt{send(\_, } \mathit{out}\texttt{, \_)} \\
\mathit{out}[0] = \mathit{SSH\_MSG\_CHANNEL\_CLOSE}
\end{array}$$

This rule means that after seeing a call to `recv()` whose second argument points to memory containing *SSH_MSG_CHANNEL_CLOSE*, we should reply with the same type of message. The full version of such a rule would also require that the reply contain the same channel identifier as the initial message.

In addition to this specification language, another key contribution of Pistachio is a novel static analysis algorithm for checking protocol implementations against their rule-based specification. Pistachio performs an *abstract interpretation* [10] to simulate the execution of program source code, keeping track of the state of program variables and of *ghost variables* representing abstract protocol state, such as the last value received in a communication. Using a fully automatic theorem prover, Pistachio checks that whenever it encounters a statement that triggers a rule (e.g., a call to `recv`), on all paths the conclusion of the rule is eventually satisfied (e.g., `send` is called with the right arguments). Although this seems potentially expensive, our algorithms run efficiently in practice because the code corresponding to a round of communication is relatively compact. Our static analysis is not guaranteed to find all rule violations, both because it operates on C, an unsafe language, and because the algorithm uses some heuristics to improve performance. In practice, however, our system missed less than 5% of known bugs when measured against a bug database.

We applied Pistachio to two benchmarks: the LSH implementation of SSH2 and the RCP implementation from Cygwin. Analysis took less than a minute for each of the test runs, and Pistachio detected a multitude of bugs in the implementations, including many security vulnerabilities. For example, Pistachio found a known problem in LSH that causes it to leak privileged information [22]. Pistachio also found a number of buffer overflows due to rule violations, although Pistachio does not detect arbitrary buffer overflows. We confirmed the bugs we found against bug databases for the projects, and we also found two new unconfirmed security bugs in LSH: a buffer overflow and an incorrect authentication failure message when using public key authentication.

In summary, the main contributions of this work are:

- We present a rule-based specification language for describing network protocol implementations. Using pattern matching to identify routines in the source code and ghost variables to track state, we can naturally represent the kinds of English specifications made in documents like RFCs. (Section 2)

- We describe a static analysis algorithm for checking that an implementation meets a protocol specification. Our approach uses abstract interpretation to simulate the execution of the program and an automatic theorem prover to determine whether the rules are satisfied. (Section 3)

- We have applied our implementation, Pistachio, to LSH and RCP. Pistachio discovered a wide variety of known bugs, including security vulnerabilities, as well as two new unconfirmed bugs. Overall Pistachio missed about 5% of known bugs and had a 38% false positive rate. (Section 4)

```
0. int main(void) {
1.   int sock, val = 1, recval;
2.   send(sock, &val, sizeof(int));
3.   while(1) {
4.     recv(sock, &recval, sizeof(int));
5.     if (recval == val)
6.       val += 2;
7.     send(sock, &val, sizeof(int));
8.   }
9. }
```

Figure 1: Simple alternating bit protocol implementation

Based on our results, we believe that Pistachio can be a valuable tool in ensuring the safety and security of network protocol implementations.

2 Rule-Based Protocol Specification

The first step in using Pistachio is developing a rule-based protocol specification, usually from a standards document. As an example, we develop a specification for a straightforward extension of the alternating bit protocol [4]. Here is a straw-man description of the protocol:

> The protocol begins with the current party sending the value $n = 1$. In each round, if n is received then the current party sends $n + 1$; otherwise the current party resends n.

Figure 1 gives a sample implementation of this protocol. Here `recv()` and `send()` are used to receive and send data, respectively. Notice that this implementation is actually flawed—on statement 6, `val` is incremented by 2 instead of by 1.

To check this protocol, we must first identify the communication primitives in the source code. In this case we see that the calls to `send()` in statements 2 and 7 and the call to `recv()` in statement 4 perform the communication. More specifically, we observe that we will need to track the value of the second argument in the calls, since that contains a pointer to the value that is communicated.

We use *patterns* to match these function calls or other expressions in the source code. Patterns contain *pattern variables* that specify which part of the call is of interest to the rule. For this protocol, we use pattern send(_, *out*, _) to bind pattern variable *out* to the second argument of `send()`, and we use pattern recv(_, *in*, _) to bind *in* to the second argument of `recv()`. For other implementations we may need to use patterns that

Rule				Description
(1)	$\emptyset$	$\Rightarrow$	`send(_, `*out*`, _)` $out[0..3] = 1$ $n := 1$	The protocol begins by sending the value 1
(2)	`recv(_, `*in*`, _)` $in[0..3] = n$	$\Rightarrow$	`send(_, `*out*`, _)` $out[0..3] = in[0..3] + 1$ $n := out[0..3]$	If n is received then send $n + 1$
(3)	`recv(_, `*in*`, _)` $in[0..3] \neq n$	$\Rightarrow$	`send(_, `*out*`, _)` $out[0..3] = n$	Otherwise resend n

Figure 2: Rule-based protocol specification

match different functions. Notice that in both of these patterns, we are already abstracting away some implementation details. For example, we do not check that the last parameter matches the size of `val`, or that the communication socket is correct, i.e., these patterns will match calls even on other sockets.

Patterns can be used to match any function calls. For example, we have found that protocol implementers often create higher-level functions that wrap send and receive operations, rather than calling low-level primitives directly. Using patterns to match these functions can make for more compact rules that are faster to check, though this is only safe if those routines are trusted.

2.1 Rule Encoding

Once we have identified the communication operations in the source code, we need to write rules that encode the steps of the protocol. Rules are of the form

$$(P_H, H) \Rightarrow (P_C, C, G)$$

where H is a *hypothesis* and C is a *conclusion*, P_H and P_C are patterns, and G is a set of assignments to ghost variables representing protocol state. In words, such a rule means: If we find a statement s in the program that matches pattern P_H, assume that H holds, and make sure that on all possible execution paths from s there is some statement matching P_C, and moreover at that point the conditions in C must hold. If a rule is satisfied in this manner, then the side effects G to ghost variables hold after the statements matching P_C.

For example, Figure 2 contains the rules for our alternating bit protocol. Rule (1) is triggered by the start of the program, denoted by hypothesis $\emptyset$. This rule says that on all paths from the start of the program, `send()` must be called, and its second argument must point to a 4-byte block containing the value 1. We can see that this rule is satisfied in statement 2 in Figure 1. As a side effect, the successful conclusion of rule (1) sets the ghost variable n to 1. Thus the value of n corresponds to the data stored in `val`. Notice that there is a call to `send()` in statement 7 that could match the conclusion pattern—but it does not, because we interpret patterns in rules to always mean the *first* occurrence of a pattern on a path.

Rule (2) is triggered by a call to `recv()`. It says that if `recv()` is called, then assuming that the value n is received in the first four bytes of in, the function `send()` must eventually be called with $in + 1$ as an argument, and as a side effect the value of n is incremented. Similarly to before, this rule matches the first occurrence of `send()` following `recv()`. In our example code this rule is not satisfied. Suppose rule (1) has triggered once, so the value of n is 1, and we trigger rule (2) in statement 4. Then if we assume n is received in statement 4, then statement 7 will send $n + 2$. Hence Pistachio signals a rule violation on this line.

Finally, rule (3) is triggered on the same call to `recv()` in statement 4. It says that if we assume the value of n is not received in in, then eventually `send()` is called with n as the argument. This rule will be satisfied by the implementation, because when we take the false branch in statement 5 we will resend `val`, which always contains n after rules (1) or (3) fire.

2.2 Developing Rule-Based Specifications

As part of our experimental evaluation (Section 4), we developed rule-based specifications for the SSH2 protocol and the RCP protocol. In both cases we started with a specification document such as an RFC or IETF standard. We then developed rules from the textual descriptions in the document, using the following steps:

1. *Identifying patterns*. The specification writer can either choose the low-level communication primitives as the primary patterns, as in Figure 2, or write rules in terms of higher-level routines. We have attempted both approaches when developing our specifications, but decided in favor of the first method for more future portability.

2. *Defining the rules*. The main task in constructing a rule-based specification is, of course, determining the basic rules. The specification writer first needs to read the standards document carefully to

discover what data is communicated and how it is represented. Then to discover the actual rules they should study the sections of the specification document that describe the protocol's behavior. We found that phrases containing *must* are good candidates for such rules. (For a discussion of terms such as *must*, *may*, and *should* in RFC documents, see RFC 1111.)

For instance, as we read the SSH2 standard we learned that the message format for SSH user authentication starts with the message type, followed by the user name, service name, and method name. Furthermore, we found that the use of "none" as the authentication method is strongly discouraged by the specification except for debugging purposes. This suggested a potential security property: To prevent anonymous users from obtaining remote shells, we should ensure that *If we receive a user authentication request containing the* none *method, we must return SSH_MSG_USERAUTH_FAILURE*. Once we determine this rule, it is easy to encode it in Pistachio's notation, given our knowledge of SSH message formats.

3. *Describing state*. Finally, as we are constructing the rules, we may discover that some protocol steps are state-based. For instance, in the SSH2 protocol, any banner message has to be sent before the server sends *SSH_MSG_USERAUTH_SUCCESS*. To keep track of whether we have sent the success message, we introduce a new ghost variable called *successSent* that is initially 0 (here we assume for simplicity that we only have one client). We modify our rules to set *successSent* to 1 or 0 in the appropriate cases. Then the condition on banner messages can be stated as *Given that successSent is 1 and for any message received, the output message type is different from SSH_MSG_USERAUTH_BANNER*. Our experience is that coming up with the ghost variables is the least-obvious part of writing a specification and requires some insight. In the LSH and RCP protocols, the state of the protocol usually depends on the previous message that was sent, and so our rules use ghost variables to track the last message.

Section 4.1 discusses the actual rule-based specifications we developed for our experiments.

3 Static Analysis of Protocol Source Code

Given a set of rules as described in Section 2 and the source code of a C program, Pistachio performs a static analysis to check that the program obeys the specified rules. Pistachio uses abstract interpretation [10] to simulate the behavior of source code. The basic idea is to associate a set of *facts* with each point during execution. In our system, the facts we need to keep track of are the predicates in the rules and anything that might be related to them. Each statement in the program can be thought of as a *transfer function* [1], which is a "fact transformer" that takes the set of facts that hold before the statement and determines the facts that hold immediately after:

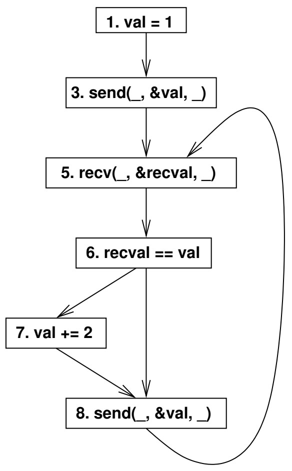
- After an assignment statement *var* = *expr*, we first remove any previous facts about *var* and then add the fact $\mathit{var} = \mathit{expr}$. For example, consider the code in Figure 1 again. If before statement 6 $\{\mathit{val} = n\}$ is the set of facts that hold, after the assignment in statement 6 the set $\{\mathit{val} = n + 2\}$ holds. Pointers and indirect assignments are handled similarly, as discussed below.

- If we see a conditional *if (p) s1 else s2*, we add the fact that p holds on the true branch, and that $\neg p$ holds on the false branch. For example, in statement 6 in Figure 1 we can always assume $\mathit{recval} = \mathit{val}$, since to reach this statement the condition in statement 5 must have been true.

- If we see a function call *f(x1, ..., xn)*, we propagate facts from the call site through the body of *f* and back to the caller. I.e., we treat the program as if all functions are inlined. As discussed below, we bound the number of times we visit recursive functions to prevent infinite looping, although recursive functions are rare in practice for network protocol implementations.

We perform our analysis on a standard *control-flow graph* (CFG) constructed from the program source code. In the CFG, each statement forms a node, and there is an edge from s_1 to s_2 if statement s_1 occurs immediately before statement s_2. For example, Figure 3(a) gives the CFG for the program in Figure 1.

Figure 4 presents our abstract interpretation algorithm more formally. The goal of this algorithm is to update *Out*, a mapping such that $\mathit{Out}(s)$ is the set of facts that definitely hold just after statement s. The input to *FactDerivation* is an initial mapping *Out*, a set of starting statements S, and a set of ending statements T. The algorithm simulates the execution of the program from statements in S to statements in T while updating *Out*. Our algorithm uses an automatic theorem prover to determine which way conditional branches are taken. In this pseudocode, we write $\mathit{pred}(s)$ and $\mathit{succ}(s)$ for the predecessor and successor nodes of s in the CFG.



(a) Control-Flow Graph

Rule (1)	**Hyp:** facts = $\emptyset$ **Concl:** stmt 2 matches, facts = $\{val = 1\}$ **Need to show:** $\{val = 1\} \wedge \{\&val = out\} \Rightarrow out[0..3] = 1$ **Action:** $n := 1$
Rule (3)	**Hyp:** stmt 4 matches, facts = $\{n = 1, val = 1, in = \&recval, in[0..3] \neq n\} = F$ **Branch:** Since assumptions $\Rightarrow$ $(recval \neq val)$, false branch taken **Concl:** stmt 7 matches, same facts F as above **Need to show:** $F \wedge \{\&val = out\} \Rightarrow out[0..3] = n$ **Action:** none
Rule (2)	**Hyp:** stmt 4 matches, facts = $\{n = 1, val = 1, in = \&recval, in[0..3] = n\}$ **Branch:** Since assumptions $\Rightarrow$ $(recval = val)$, true branch taken **Concl:** stmt 7 matches, facts are $F = \{n = 1, val = 3, in = \&recval, in[0..3] = n\}$ **Need to show:** $F \wedge \{\&val = out\} \Rightarrow (out[0..3] = in[0..3] + 1)$ **Fails** to hold; issue warning

(b) Algorithm Trace

Figure 3: Static Checking of Example Program

Our simulation algorithm uses a worklist Q of statements, initialized on line 1 of Figure 4. We repeatedly pick statements from the worklist until it is empty. When we reach a statement in T on line 6, we stop propagation along that path. Because the set of possible facts is large (most likely infinite), simulation might not terminate if the code has a loop. Thus on line 10 we heuristically stop iterating once we have visited a statement *max_pass* times, where *max_pass* is a predetermined constant bound. Based on our experiments, we set *max_pass* to 75. We settled on this value empirically by observing that if we vary the number of iterations, then the overall false positive and false negative rates from Pistachio rarely changed after 75 iterations in our experiments.

In line 5 of the algorithm, we compute the set *In* of facts from the predecessors of s in the CFG. If the predecessor was a conditional, then we also add in the appropriate guard based on whether s is on the true or false branch. Then we apply a transfer function that depends on what kind of statement s is: Lines 13–15 handle simple assignments, which kill and add facts as described earlier, and then add successor statements to the worklist. Lines 16–24 handle conditionals. Here we use an automatic theorem prover to prune impossible code paths. If the guard p holds in the current state, then we only add s_1 to the worklist, and if $\neg p$ holds then we only add s_2 to the worklist. If we cannot prove either, i.e., we do not know which path we will take, then we add both s_1 and s_2 to the worklist. Finally, lines 25–32 handle function calls. We compute a renaming *map* between the actual and formal parameters of f, and then recursively simulate f from its entry node to its exit nodes. We start simulation in state $map(Out)$, which contains the facts in *Out* renamed by *map*. Then the state after the call returns is the intersection of the states at all possible exits from the function, with the inverse mapping map^{-1} applied.

C includes a number of language features not covered in Figure 4. Pistachio uses CIL [26] as a front-end, which internally simplifies many C constructs by introducing temporary variables and translating loops into canonical form. We unroll loops up to *max_pass* times in an effort to improve precision. However, as discussed in Section 3.2, we attempt to find a fixpoint during unrolling process and stop if we can do so, i.e., if we can find a loop invariant. C also includes pointers and a number of unsafe features, such as type casts and unions. Pistachio tracks facts about pointers during its simulation, and all C data is modeled as arrays of bytes with bounds information. When there is an indirect assignment through a pointer, Pistachio only derives a fact if the theorem prover can show that the write is within bounds, and otherwise kills all existing facts about the array. Note that even though a buffer overflow may modify other memory, we do not kill other facts, which is unsound but helps reduce false positives. Since all C data is represented as byte arrays, type casts are implicitly handled as well, as long we can determine at analysis time the allocation size for each type, which Pistachio could always do in our experiments. In addition, in order to reduce false positives Pistachio assumes that variables are initialized with their default values independently of their scope. In the next sections, we illustrate the use of *FactDerivation* during the process of checking the alternating bit protocol from Figure 2.

```
FactDerivation(Out, $S$, $T$)
1: $Q \leftarrow S$
2: while $Q$ not empty do
3:     $s \leftarrow \mathit{dequeue}(Q)$
4:     $\mathit{visit}(s) \leftarrow \mathit{visit}(s) + 1$
5:     $\mathit{In} \leftarrow \bigcap_{s' \in \mathit{pred}(s)} \begin{cases} \mathit{Out}(s') \cup \{C\} & s' \text{ is “if}(C) \text{ then } s \text{ else } s_2\text{”} \\ \mathit{Out}(s') \cup \{\neg C\} & s' \text{ is “if}(C) \text{ then } s_1 \text{ else } s\text{”} \\ \mathit{Out}(s') & \text{otherwise} \end{cases}$
6:     if $s \in T$ then
7:         $\mathit{Out}(s) \leftarrow \mathit{In}$
8:         continue
9:     end if
10:    if $\mathit{visit}(s) > \mathit{max\_pass}$ then
11:        continue
12:    end if
13:    if $s$ is assignment “var=expr” then
14:        $\mathit{Out}(s) \leftarrow (\mathit{In} - \{\text{facts involving } \mathit{var}\}) \cup \{\mathit{var}{=}\mathit{expr}\}$
15:        $Q \leftarrow Q \cup \mathit{succ}(s)$
16:    else if $s$ is “if($p$) then $s_1$ else $s_2$” then
17:        $\mathit{Out}(s) \leftarrow \mathit{In}$
18:        if $\mathit{Theorem\text{-}prover}(\mathit{Out}(s) \Rightarrow p) = \mathit{yes}$ then
19:            $Q \leftarrow Q \cup \{s_1\}$
20:        else if $\mathit{Theorem\text{-}prover}(\mathit{Out}(s) \Rightarrow \neg p) = \mathit{yes}$ then
21:            $Q \leftarrow Q \cup \{s_2\}$
22:        else
23:            $Q \leftarrow Q \cup \{s_1, s_2\}$
24:        end if
25:    else if $s$ is “$f(x_1, \ldots, x_n)$” then
26:        map ← mapping between actual and formal parameters
27:        $\mathit{start}' \leftarrow$ entry statement of $f$
28:        $T' \leftarrow$ exit statements of $f$
29:        $\mathit{FactDerivation}(\mathit{map}(\mathit{Out}), \{\mathit{start}'\}, T')$
30:        $\mathit{Out}(s) \leftarrow \bigcap_{s' \in T'} \mathit{map}^{-1}(\mathit{Out}(s'))$
31:        $Q \leftarrow Q \cup \mathit{succ}(S)$
32:    end if
33: end while
```

Figure 4: Fact derivation in Pistachio

3.1 Checking a Single Rule

Given the *FactDerivation* algorithm, we can now present our algorithm for checking that the code obeys a single rule R of the form $(P_H, H) \Rightarrow (P_C, C, G)$. Assume that we are given a set of statements S that match P_H. Then to check R, we need to simulate the program forward from the statements in S using *FactDerivation*. We check that we can reach statements matching P_C along all paths and that the conclusion C holds at those statements. Figure 5 gives our formal algorithm *CheckSingleRule* for carrying out this process.

The input to *CheckSingleRule* is a rule R, an initial set of facts *Out*, and a set of starting statements S. For all statements in S, on line 3 we add to their facts the assumptions H and any facts derived from pattern-matching S against P_H; we denote this latter set of facts by $P_H(s)$. If there is some path from S along which we cannot find a match to P_C, we report an error on line 5 and consider the rule unsatisfied. Otherwise on line 6 we search forward along all program paths until we first find the conclusion pattern P_C. Then on line 7 we perform abstract interpretation using *FactDerivation* to update *Out*. On line 8, we use the theorem prover to check whether the conclusion C holds at the statements that match P_C. If they do then the rule is satisfied, and lines 11–12 update *Out*(s') with facts for ghost variables. We also remove any facts about pattern variables (*in* and *out* in our examples) from *Out* (line 14).

```
CheckSingleRule($R$, Out, $S$)
1: Let $R$ be of the form $(P_H, H) \Rightarrow (P_C, C, G)$
2: for $s \in S$ do
3:     $\mathit{Out}(s) \leftarrow \mathit{Out}(s) \cup H \cup P_H(s)$
4: end for
5: If there is a path from $S$ on which no statement matches $P_C$, return error
6: Let $T$ be the set of statements that are the first matches to $P_C$ on all paths
   from statements in $S$
7: $\mathit{FactDerivation}(\mathit{Out}, S, T)$
8: if $\forall t \in T$, $\mathit{Theorem\text{-}prover}((\mathit{Out}(t) \wedge P_C(t)) \Rightarrow C) = \mathit{yes}$ then
9:     /* $R$ is satisfied */
10:    for $s' \in T$ do
11:        Remove from $\mathit{Out}(s')$ facts involving ghost variables modified in $G$
12:        $\mathit{Out}(s') \leftarrow \mathit{Out}(s') \cup G$
13:    end for
14:    Remove from Out all the facts involving pattern variables
15:    Return rule satisfied
16: else
17:    /* $R$ is not satisfied */
18:    Return error
19: end if
```

Figure 5: Algorithm for checking a single rule

We illustrate using *CheckSingleRule* to check rule (1) from Figure 2 on the code in Figure 1. The first block in Figure 3(b) lists the steps taken by the algorithm. We will discuss the remainder of this figure in Section 3.2.

In rule (1), the hypothesis pattern P_H is the start of the program, and the set of assumptions H is empty. The conclusion C of this rule is $out[0..3] = 1$, where out matches the second argument passed to a call to `send()`. Thus to satisfy this rule, we need to show that $out[0..3] = 1$ at statement 2 in Figure 1. We begin by adding H and $P_H(0)$, which in this case are empty, to *Out*(0), the set of facts at the beginning of the program, which is also empty. We trace the program from this point forward using *FactDerivation*. In particular, $\mathit{Out}(1) = \mathit{Out}(2) = \{\mathit{val} = 1\}$. At statement 2 we match the call to `send()` against P_C, and thus we also have fact $\&\mathit{val} = \mathit{out}$. Then we ask the theorem prover to show $\mathit{Out}(2) \wedge \{\&\mathit{val} = \mathit{out}\} \Rightarrow C$. In this case the proof succeeds, and so the rule is satisfied, and we set ghost variable n to 1.

3.2 Checking a Set of Rules

Finally, we develop our algorithm for checking a set of rules. Consider again the rules in Figure 2. Notice that rules (2) and (3) both depend on n, which is set in the conclusion of rules (1) and (2). Thus we need to check

```
CheckRuleSet($W$, Out, $S$)
 1: while $W$ not empty do
 2:     $R \leftarrow dequeue(W)$
 3:     $visit(R) \leftarrow visit(R) + 1$
 4:     if $visit(R) > max\_pass$ then
 5:         continue
 6:     end if
 7:     Let $R$ be of the form $(P_H, H) \Rightarrow (P_C, C, G)$
 8:     Let $T$ be the set of statements that are the first matches to $P_H$ on all paths
        from statements in $S$
 9:     Let $Out' = Out$
10:     $FactDerivation(Out', S, T)$
11:     $CheckSingleRule(R, Out', T)$
12:     Let $U$ be the set of statements computed in step 6 in Figure 5
13:     if $R$ is satisfied then
14:         CheckRuleSet($\{R' \mid R'$ depends on $R\}, Out', U)$
15:     end if
16: end while
```

Figure 6: Algorithm for checking a set of rules

whether rules (2) or (3) are triggered on any program path after we update n, and if they are, then we need to check whether they are satisfied. Since rule (2) depends on itself, we in fact need to iterate. Formally, we say that rule R_i *depends on* rule R_j if R_j sets a ghost variable that R_i uses in its hypothesis. We can think of dependencies as defining a graph between rules, and we use a modified depth-first search algorithm to check rules in the appropriate order based on dependencies.

Figure 6 gives our algorithm for checking a set of rules. The input is a set of rules W that need to be checked, a mapping *Out* of facts at each program point, and a set of statements S from which to begin checking the rules in W. To begin checking the program, we call *CheckRuleSet*(R_0, Out_0, S_0), where R_0 is the rule with hypothesis $\emptyset$ (we can always create such a rule if it does not exist), S_0 is the initial statement in the program, and Out_0 maps every statement to the set of all possible facts.

Then the body of the algorithm removes a rule R from the worklist W and checks it. Because rule dependencies may be cyclic, we may visit a rule multiple times, and as in Figure 5 we terminate iteration once we have visited a rule *max_pass* times (line 5). On line 8 we trace forward from S to find all statements T that match P_H. Then we copy the current set of facts *Out* into a new set *Out*$'$, simulate the program from S to T (line 10), and then on line 11 check the rule R with facts *Out*$'$. Notice that the call to *FactDerivation* on line 10 and the call to *CheckSingleRule* on line 11 modify the copy *Out*$'$ while leaving the original input *Out* unchanged. This means that in the next iteration of the loop, when we pick another rule from the worklist and check it starting from S, we will again check it using the initial facts in *Out*, which is the intended behavior: A call to *CheckRuleSet* should check all rules in W starting in the same state. Finally, on line 14, if the rule R was satisfied, we compute the set of all rules that depend on R, and then we recursively check those rules, starting in our new state from statements that matched the conclusion pattern.

We illustrate the algorithm by tracing through the remainder of Figure 3(b), which describes the execution of *CheckRuleSet* on our example program. Observe that rule (1) can be checked with no assumptions, hence we use it to begin the checking process. We begin with the initial statement in the program and call *CheckSingleRule*. As described in Section 3.1, we satisfy rule (1) at statement 2, and we set ghost variable n to 1. Thus after checking rule (1), we can verify rules that depend on n's value. For our example, either rule (2) or rule (3) might fire next, and so W will contain both of them.

Checking rule (3). Suppose we next choose rule (3). We continue from statement 2, since that is where we set ghost variable n, and we perform abstract interpretation forward until we find a statement that matches the hypothesis pattern of rule (3), which is statement 4. Now we add the hypothesis assumption $in[0..3] \neq n$ to our set of facts and continue forward. When we reach statement 5, the theorem prover shows that the false branch will be taken, so we only continue along that one branch—which is important, because if we followed the other branch we would not be able to prove the conclusion. Taking the false branch, we fall through to statement 7, and the theorem prover concludes that rule (3) holds.

Checking rule (2). Once we are done checking rule (3), we need to go back and start checking rule (2) where we left off after rule (1), namely just after we set $n := 1$ after statement 2. The set of facts contains $n = 1$ and $val = 1$, both set during the checking of rule (1). We continue forward from statement 2, match the hypothesis of rule (2) at statement 4, and then this time at the conditional we conclude that the true branch is taken, hence *val* becomes 3 at statement 7. Now the conclusion of the rule cannot be proven, so we issue a warning that the protocol was violated at this statement.

Finding a fixpoint. Suppose for illustration purposes that rule (2) had checked successfully, i.e., in statement 6, `val` was only incremented by one. Then at statement 7 we would have shown the conclusion and set ghost variable $n := out[0..3]$. Then since we changed the value of n, we need to re-check rules (2) and (3) starting from this point, since they depend on the value of n. Of course, if we follow the loop around again we would need to check rules (2) and (3) yet again, leading to an infinite loop, cut off after *max_pass* times.

In order to model this case more efficiently, Pistachio's implementation of *CheckRuleSet* includes a technique for finding a fixpoint and safely stopping abstract interpretation early. Due to lack of space, we omit pseudocode for this technique and illustrate it by example. During the first check of rule (2), we would have that $\mathit{Out}^0(2) = \{n = \mathit{val}, \mathit{val} = 1\}$, where the superscript of *Out* indicates how many iterations we have performed. After rule (2) succeeds at statement 7, we set n to *val*, and hence $\mathit{Out}^1(7) = \{n = \mathit{val}, \mathit{val} = 2\}$. Since rule (2) depends on itself, we need to check it once again. After simulating the loop in statements 3–8 another time, we would check rule (2) with facts $\mathit{Out}^2(7) = \{n = \mathit{val}, \mathit{val} = 3\}$. Rather than continuing until we reach *max_pass*, we instead try to intersect the facts we have discovered to form a potential loop invariant. We look at the set of facts that hold just before rule (2) is triggered and just after rule (2) is verified:

$\mathit{Out}^0(2) = \{n = \mathit{val}, \mathit{val} = 1\}$	Initial entry to loop
$\mathit{Out}^1(7) = \{n = \mathit{val}, \mathit{val} = 2\}$	Back-edge from the first iteration
$\mathit{Out}^2(7) = \{n = \mathit{val}, \mathit{val} = 3\}$	Back-edge from the second iteration

Generally speaking, at step k we are interested in the intersection $\mathit{Out}^0(2) \cap \bigcap_{i\in[1,k]} \mathit{Out}^i(7)$. In our case, the only fact in this intersection is $n = \mathit{val}$. We then set $\mathit{Out}^{k+1} = \{n = \mathit{val}\}$ (thus ignoring the particular value of *val* for step k) and attempt to verify rule (2) once more. Rule (2) indeed verifies, which means we have reached a fixpoint, and we can stop iterating well before reaching *max_pass*. We found this technique for finding fixpoints effective in our experiments (Section 4).

4 Implementation and Experiments

Our tool Pistachio is implemented in approximately 6000 lines of OCaml. We use CIL [26] to parse C programs and Darwin [5] as the automated theorem prover. Darwin is a sound, fully-automated theorem prover for first order clausal logic. We chose Darwin because it performs well and can handle the limited set of Horn-type theorems we ask it to prove. Since Darwin is not complete, it can return either "yes," "no," or "maybe" for each theorem. Pistachio conservatively assumes that a warning should be generated if a rule conclusion cannot be provably implied by the facts derived starting from the hypothesis.

In order to analyze realistic programs, Pistachio needs to model system libraries. When Pistachio encounters a library call, it generates a skeleton rule-based specification for the function that can be filled in by the user. Rules for library functions are assumed correct, rather than checked. For our experiments we added such specifications for several I/O and memory allocation routines. There are some library functions we cannot fully model in our framework, e.g., $\mathit{geterrno}()$, which potentially depends on any other library call. In this case, Pistachio assumes that conditionals based on the result of $\mathit{geterrno}()$ may take either branch, which can reduce precision.

4.1 Core Rule Sets

We evaluated Pistachio by analyzing the LSH [21] implementation of SSH2 and the RCP implementation from Cygwin's inetutils package. We chose these systems because of their extensive bug databases and the number of different versions available.

We created rule-based specifications by following the process described in Section 2.2. Our (partial) specification for SSH2 totaled 96 rules, and the one for RCP totaled 58 rules. Because the rules describe particular protocol details and are interdependent, it is difficult to correlate individual rules with general correctness or security properties. In Figure 7, we give a rough breakdown of the rules in our SSH2 specification, and we list example rules with descriptions. These rules are taken directly from our experiments—the only changes are that we have reformatted them in more readable notation, rather than in the concrete grammar used in our implementation, and we have used `send` and `recv` rather than the actual function names.

The categories we chose are based on functional behavior. The first category, *message structure and data transfer*, includes rules that relate to the format, structure, and restrictions on messages in SSH2. The example rule requires that any prompt sent to keyboard interactive clients not be the empty string. The second category, *compatibility rules*, includes rules about backwards compatibility with SSH1. The example rule places a requirement on sending an identification string in compatibility mode. The third category, *functionality rules*, includes rules about what abilities should or should not be supported. The example rule requires that the implementation not allow the "none" authentication method. The last category, *protocol logic rules*, contains the majority of the rules. These rules require that the proper response is sent for each received message. The first example rule requires that the server provide an adequate

Category	Example rule(s)	Description
Message structure and data transfer	$\texttt{recv}(\textit{in\_sock}, \textit{in}, \_)$ $\textit{in.msgid} = \textit{SSH\_MSG\_USERAUTH\_REQUEST}$ $\textit{in.authtype} = \textit{“keyboard-interactive”}$ $\Rightarrow$ $\texttt{send}(\textit{out\_sock}, \textit{out}, \_)$ $\textit{in\_sock} = \textit{out\_sock}$ $\textit{out.msgid} = \textit{SSH\_MSG\_USERAUTH\_INFO\_REQUEST}$ $\textit{len}(\textit{out.prompt}) > 0$	In keyboard interactive authentication mode, the prompt field(s) [the server sends to the interactive client] MUST NOT be empty.
Compatibility	$\texttt{recv}(\textit{sock}, \textit{in}, \_)$ $\textit{in}[\textit{len}(\textit{in}) - 2] = \textit{CR}$ $\textit{in}[\textit{len}(\textit{in}) - 1] = \textit{LF}$ $\textit{connected}[\textit{sock}] = 0$ $\Rightarrow$ $\texttt{send}(\textit{sock}, \textit{out}, \_)$ $\textit{out}[\textit{len}(\textit{out}) - 2] = \textit{CR}$ $\textit{out}[\textit{len}(\textit{out}) - 1] = \textit{LF}$ $\textit{connected}[\textit{sock}] := 1$	In compatibility mode, after receiving the client identification string, the server MUST NOT send any additional messages before its identification string. [Note: The message with the identity string is distinguished by ending in a CR/LF combination]
Functionality	$\texttt{recv}(\_, \textit{in}, \_)$ $\textit{in}[0] = \textit{SSH\_MSG\_USERAUTH\_REQUEST}$ $\textit{isOpen}[\textit{in}[1..4]] = 1$ $\textit{in}[21..25] = \textit{“none”}$ $\Rightarrow$ $\texttt{send}(\_, \textit{out}, \_)$ $\textit{out}[0] = \textit{SSH\_MSG\_USERAUTH\_FAILURE}$	It is STRONGLY RECOMMENEDED that the “none” authentication method not be supported.
Protocol logic	$\texttt{recv}(\textit{in\_sock}, \textit{in}, \_)$ $\textit{in}[0] = \textit{SSH\_MSG\_GLOBAL\_REQUEST}$ $\textit{in}[1..14] = \textit{“tcpip-forward”}$ $\textit{in}[15] = 1$ $\textit{in}[(\textit{len}(\textit{in}) - 4)..(\textit{len}(\textit{in}) - 1)] = 0$ $\Rightarrow$ $\texttt{send}(\textit{out\_sock}, \textit{out}, \_)$ $\textit{in\_sock} = \textit{out\_sock}$ $\textit{out}[0] = \textit{SSH\_MSG\_REQUEST\_SUCCESS}$	The server MUST respond to a TCP/IP forwarding request in which the *wantreply* flag [byte 15] is set to 1 and the *port* [last 4 bytes] is set to 0 with a *SSH_MSG_REQUEST_SUCCESS* containing the forwarding port.
	$\texttt{recv}(\textit{in\_sock}, \textit{in}, \_)$ $\textit{in}[0] = \textit{SSH\_MSG\_GLOBAL\_REQUEST}$ $\textit{in}[15] = 1$ $\Rightarrow$ $\texttt{send}(\textit{out\_sock}, \textit{out}, \_)$ $\textit{in\_sock} = \textit{out\_sock}$	The server MUST respond to all global requests with the *wantreply* flag [byte 15] set to 1.

Figure 7: Categorization and example rules for the SSH2 protocols

response to TCP/IP forwarding requests with a value of 0 for *port*. The second rule requires that the server replies to all global requests that have the *wantreply* flag set.

Based on our experience developing rules for SSH and RCP, we believe that the process does not require any specialized knowledge other than familiarity with the rule-based language grammar and the specification document. It took the authors less than 7 hours to develop each of our SSH2 and RCP specifications. The rules are generally slightly more complex than is shown in Figure 7, containing on average 11 facts in the hypothesis and 5 facts in the conclusion for LSH, and 9 facts for the hypothesis and 4 for the conclusion for RCP. Originally, we started with a rule set derived directly from specification documents, which was able to catch many but not all bugs, and then we added some additional rules (a little over 10% more per specification) to look for some specific known bugs (Section 4.4). These additional rules produced about 20% of the warnings from Pistachio. Generally, the rules written from the specification document tend to catch missing functionality and message format related problems, but are less likely to catch errors relating to compatibility problems or unsafe use of library functions. The process of extending the initial set of rules proved fairly easy, since specific information about the targeted bugs was available from the respective bug databases.

4.2 Results for Core Rule Sets

We started with initial specifications for the SSH2 protocols (87 rules) and the RCP protocol (51 rules), with “core rules” based only on specification documents. In Section 4.4 we discuss extending these rules to target particular bugs. Using these specifications, we ran our tool against several different versions of LSH, ranging from 0.1.3 to 2.0.1, and several different versions of RCP, ranging from 0.5.4 to 1.3.2. We used the same specification for all versions of the code, and we ran Pistachio with *max_pass* set to 75.

Figure 8 presents the results of our analysis. We list the lines of code (omitting comments and blank lines) analyzed by Pistachio. We only include code involved in the rules, e.g., we omit the bodies of functions our rules

Version	**0.1.3**	**0.2.1**	**0.2.9**	**0.9.1**	**1.0.1**	**1.1.1**	**1.2.1**	**1.3.1**	**1.4.1**	**1.5.1**	**1.5.5**	**2.0.1**
Analyzed code size (lines)	8745	8954	9145	9267	9221	10431	10493	10221	12585	12599	12754	13689
Running time (s)	23.96	25.24	24.99	25.89	25.21	26.85	28.91	30.1	29.74	31.45	36.3	38.91
Bugs in database	91	83	69	65	81	80	82	82	51	40	31	13
Warnings from Pistachio	118	123	110	97	91	93	92	82	78	74	33	25
False positives	40	57	43	43	22	27	24	16	38	38	7	12
False negatives	5	5	3	4	3	3	2	2	1	1	0	1

(a) LSH implementations

Version	**0.5.4**	**0.6.4**	**0.8.4**	**1.1.4**	**1.2.3**	**1.3.2**
Analyzed code size (lines)	4501	4723	4813	4865	4891	5026
Running time (s)	16.13	18.34	19.13	18.56	21.65	25.43
Bugs in database	51	46	47	51	28	23
Warnings from Pistachio	73	89	58	70	48	39
False positives	30	30	17	25	27	19
False negatives	2	2	1	2	0	1

(b) RCP implementations

Figure 8: Analysis Results with Core Rule Sets

treat as opaque, such as cryptographic functions.

The third row contains the running time of our system. (The running times include the time for checking the core rules plus the additional rules discussed in the next section.) In all cases the running times were under one minute. The next four rows measure Pistachio's effectiveness. We list the number of bugs found in the project's bug database for that version and the number of warnings issued by Pistachio. We counted only bugs in the database that were reported for components covered by our specifications. For instance, we only include bugs in LSH's authentication and transport protocol code and not in its connection protocol code. We also did not include reports that appeared to be feature requests or other issues in our bug counts. The last two rows report the number of false positives (warnings that do not correspond to bugs in the code) and false negatives (bugs in the database we did not find).

We found that most of the warnings reported by Pistachio corresponded to bugs that were in the database. We also found two apparently new bugs in LSH version 2.0.1. The first involves a buffer overflow in buffers reserved for passing environment variables in the implementation of the SSH Transport protocol. The second involves an incorrectly formed authentication failure message when using PKI. We have not yet confirmed these bugs; we sent an email to the LSH mailing list with a report, but it appears that the project has not been maintained recently, and we did not receive a response.

To determine whether a warning is a bug, we trace the fact derivation process in reverse, using logs produced by Pistachio that give *Out* for each analyzed statement. We start from a rule failure at a conclusion, and then examine the preceding statements to see whether they directly contradict those facts. If so, then we have found what we think is a bug, and we look in the bug database to judge whether we are correct. If the preceding statements do not directly contradict the facts, we continue tracing backward, using *Out* to help understand the results, until we either find a contradiction or reach the hypothesis, in which case we have found a false positive. If the conclusion pattern is not found on all program paths, we use the same process to determine why those paths were not pruned from the search during fact derivation.

```
// Code is extracted from a function handling connection initialization
......
1. fmsgrecv(clisock,inmsg,SSH2_MSG_SIZE);
2. if(!parse_message(MSGTYPE_PROTOVER,inmsg,len(inmsg),&protomsg))
3.      return;
......
4. if(protomsg.proto_ver < 1) {
5.      payload.msgid = SSH_DISCONNECT;
6.      payload.reason = SSH_DISCONNECT_PROTOCOL_ERROR;
......
7.      sz = pack_message(MSGTYPE_DISCONNECT,payload,
                         outmsg,SSH2_MSG_SIZE);
8. } else {
9.      sprintf(outstr,"%.1f%c%s%c%c",2,SP,SRV_COMMENTS,CR,LF);
10.     sz = pack_message(MSGTYPE_PROTOVER,outstr,
                         outmsg,SSH2_MSG_SIZE);
......
11. }
12. fmsgsend(clisock,outmsg,sz);
```

Figure 9: Sample compatibility bug

As a concrete example of this process, consider the second rule in Figure 7. This rule is derived from section 5 of the SSH2 Transport Protocol specification, and Pistachio reported that this rule was violated for LSH implementation 0.2.9, since it could not prove that $out[len(out) - 2] = CR$ and $out[len(out) - 1] = LF$ at

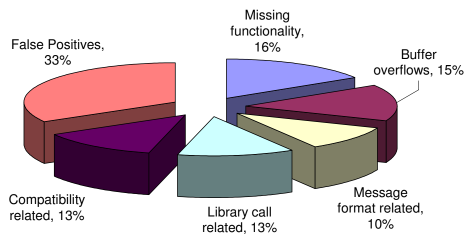


Figure 10: Error breakdown by type in LSH version 1.2.1

```
1. fmsgrecv(clisock,inmsg,SSH2_MSG_SIZE);
2. if(!parse_message(MSGTYPE_USERAUTHREQ,inmsg,len(inmsg),&authreq))
3.      return;
......
4. if(authreq.method==USERAUTH_PKI) {
......
5. } else if(authreq.method==USERAUTH_PASSWD) {
......
6. } else {
......
7. }
8. sz = pack_message(MSGTYPE_REQSUCCESS,payload,
                    outmsg,SSH2_MSG_SIZE);
9. fmsgsend(clisock,outmsg,sz);
```

Figure 11: Sample functionality bug

a conclusion. Figure 9 shows the code (slightly simplified for readability) where the bug was reported. Statement 12 matches the conclusion pattern (here *fmsgsend* is a wrapper around *send*), and so that is where we begin. Statement 12 has two predecessors in the CFG, one for each branch of the conditional statement 4. Looking through the log, we determined that the required facts to show the conclusion were in $Out(10)$ but not in $Out(7)$. We examined statement 7, and observed that if it is executed (i.e., if the conditional statement 4 takes the true branch), then line 10 will send the a disconnect message, which is incorrect. Thus we backtracked to statement 4 and determined that *protomsg.proto_ver<1* could not be proved either true or false based on $In(4)$, which was correctly derived from the hypothesis (asserted in $Out(1)$). Thus we determined that we found a bug, in which the implementation could send an *SSH_DISCONNECT* message for clients with versions below 1.0, although the protocol specifies that the server must send the identification first, independently of the other party's version. We then confirmed this bug against the LSH bug database.

While it is non-trivial to determine whether the reports issued by Pistachio correspond to bugs, we found it was generally straightforward to follow the process described above. Usually the number of facts we were trying to trace was small (at most 2-3), and the number of preceding statements was also small (rarely larger than 2). In the vast majority of the cases, it took on the order of minutes to trace through a Pistachio warning, though in some cases it took up to an hour (often due to insufficiently specified library functions that produced many paths). In general, the effort required to understand a Pistachio warning is directly proportional to the complexity of the code making up a communication round.

Figure 10 gives a more detailed breakdown of our results on LSH version 1.2.1. We divide the warnings and bugs into five main categories. The categories mostly but do not completely correspond to those in Figure 7.

Functionality errors correspond to missing functionality for which the implementation does not degrade gracefully, or conversely, to additional functionality that should not be present. The vast majority of these bugs were found as violations of rules in the *functionality* category from Figure 7. For example, the third rule in Figure 7 detected a functionality bug in LSH version 0.1.3, for the code in Figure 11. In this case, a message is received at statement 2, and Pistachio assumes the rule hypotheses, which indicate that the message is a user authorization request. Then a success message is always sent in statement —but the rule specifies that the "none" authentication method must result in a failure response. Tracing back through the code, we discovered that the **else** statement on line 6 allows the "none" method to succeed. This statement should have checked for the "hostbased" authentication method, and indeed this corresponds to a bug in the LSH bug database.

Message format errors correspond to problems with certain message formats, and are found by violations of the *message format and data transfer* rules. For example, the first rule in Figure 7 detected a violation in LSH version 0.2.9 (code not shown due to lack of space). In this version of LSH, the server stores the string values of the possible prompts in an array terminated by the empty string. However, the implementation uses the array size and not the empty string terminator to determine the end of the array, which causes the implementation to potentially include the empty string terminator as one of the prompts, violating the rule.

Compatibility related errors correspond to problems in the implementation that cause it to work incorrectly with clients or servers that implement earlier versions of the SSH protocol. These bugs correspond to violations of *compatibility* rules, and the earlier discussion of the code in Figure 9 illustrated one example.

Buffer overflows are detected by Pistachio indirectly

```
0.  char laddr[17]; int lport;
......
1. fmsgrecv(clisock,inmsg,SSH2_MSG_SIZE);
2. if(!parse_message(MSGTYPE_GLOBALREQ,inmsg,len(inmsg),&globreq))
3.      return;
......
4. if(globreq.msgtype=MSGSUBTYPE_TCPIPFORWARD) {
5.      strcpy(laddr,getstrfield(globreq.payload,0));
6.      lport = getuint32field(globreq.payload,1);
......
7.      if(!create_forwarding(clisock, laddr,lport))
8.        return debug_error();
9.      if((globreq.wantreply==1) && (lport == 0)) {
10.       payload.msgid = SSH_REQUEST_SUCCESS;
11.       payload.reason=lport;
12.       sz = pack_message(MSGTYPE_REQSUCCESS,payload,
                      outmsg,SSH2_MSG_SIZE);
13.       fmsgsend(clisock,outmsg,sz);
14.     }
15.}
```

Figure 12: Sample buffer overflow

```
1. fmsgrecv(clisock,inmsg,SSH2_MSG_CHANNEL_REQUEST);
2. if(!parse_message(MSGTYPE_CHREQ,inmsg,len(inmsg),&chreq))
3.     return;
......
4. if(chreq.msgtype==MSGSUBTYPE_SHELL) {
......
     /* fmod was previously set to "rw" */
5.     if(!(clish = popen(make_clishell(clisock),fmod)))
6.         return debug_error();
......
```

Figure 13: Sample library call bug

during rule checking. Recall that when Pistachio sees a write to an array that it cannot prove is in-bounds, then it kills facts about the array. Thus sometimes when we investigated why a conclusion was not provable, we discovered it was due to an out-of-bounds write corresponding to a buffer overflow. For example, consider the code in Figure 12, which is taken from LSH version 0.9.1 (slightly simplified). While checking this code, we found a violation of the fourth rule in Figure 7, as follows. At statement 1, Pistachio assumes the hypothesis of this rule, including that the *wantreply* flag (corresponding to *in*[15]) is set, and that the message is for TCP forwarding. Under these assumptions, Pistachio reasons that the true branch of statement 4 is taken. But then line 5 performs a *strcpy* into laddr, which is a fixed-sized locally-allocated array. The function *getstrfield()* (not shown) extracts a string out of the received message, but that string may be more than 17 bytes. Thus at the call to *strcpy*, there may be a write outside the bounds of laddr, and so we kill facts about laddr. Then at statement 7, we call *create_forwarding()*, which expects laddr to be null-terminated—and since we do not know whether there is a null byte within laddr, Pistachio determines that *create_forwarding()* might return false, causing us to return from this code without executing the call to *fmsgsend* in statement 13.

In this case, if Pistachio had been able to reason at statement 5 that laddr was null-terminated, then it would not have issued a warning. Although the return statement 8 might seem be reachable in that case, looking inside of *create_forwarding()*, we find that can only occur if LSH runs out of ports, and our model for library functions assumes that this never happens. (Even if an ill-formed address is passed to *create_forwarding()*, it still creates the forwarding for 0.0.0.0.) On the hand, if *create_forwarding()* had relied on the length of laddr, rather than it being null-terminated, then Pistachio would not have reported a warning here—even though there still would be a buffer overflow in that case. Thus the ability to detect buffer overflows in Pistachio is somewhat fragile, and it is a side effect of the analysis that they can be detected at all. Buffer overflows that do not result in rule violations, or that occur in code we do not analyze, will go unnoticed.

Library call errors correspond to unsafe use of library functions. These bugs are generally found the same ways buffer overflows are, as a side effect of rule checking. For example, Figure 13 contains code from LSH 0.9.1 that violated the last rule in Figure 7. In this case, we matched the hypothesis of the rule at statement 1, and then matched the request type at statement 2, and thus one possible path leads to the call to *popen* in statement 5. In this case, our model of *popen* requires that the second argument must be either "r" or "w," or the call to *popen* yields an undefined result. Since before statement 5 fmod was set to "rw," Pistachio assumes that *popen* may return any value, including null, and thus statement 6 may be executed and return without sending a reply message, thus violating the rule conclusion. Note that our model of *popen* always succeeds if valid arguments are passed, and thus if fmod were "r" or "w" a rule violation would not have been reported.

In addition to warnings that correspond to bugs, Pistachio also issues a number of false positives. Figure 14 breaks down the causes of false positives found in LSH, averaged over all versions. The main cause of false positives is insufficient specification of library calls. This is primarily due to the fact that library functions sometimes rely on external factors such as system state (e.g., whether *getenv()* returns NULL or not depends on which environment variables are defined) that cannot be fully modeled using our rule-based semantics. For such functions, only partial specifications can be devised. The remaining false positives are due to limitations of the theorem prover and to loop breaking, where we halt iteration

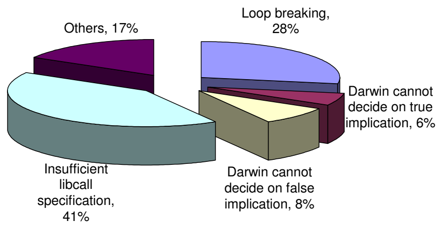


Figure 14: Causes for false positives in LSH

of our algorithm after *max_pass* times.

Besides false positives, Pistachio also has false negatives, as measured against the LSH and RCP bug databases. From our experience, these are generally caused by either assumptions made when modeling library calls, or by the fact that the rule sets are not complete. As an example of the first case, we generally make the simplifying assumption that on `open()` calls, there are sufficient file handles available. This caused a false negative for LSH version 0.1.3, where a misplaced `open()` call within a loop lead to the exhaustion of available handles for certain SSH global requests.

4.3 Security Implications

As can be seen from the previous discussion, many of the bugs found by Pistachio have obvious security implications. Even bugs that initially appear benign may introduce security vulnerabilities, depending on what constitutes a vulnerability in a particular circumstance. However, in order to measure our results we looked through the bug databases to identify which of the bugs are either clearly security-related by their nature or were documented as security-related. On average, we classified approximately 30% of the warnings (excluding false positives) as security-related for LSH and approximately 23% for RCP. Of these, buffer overflows account for approximately 53% of the security-related bugs. We consider all buffer overflows security-related. Access control warnings account for 20% of the total. These refer to the execution of functions for which the user does not have sufficient privileges. Finally, compatibility problems account for 18% of the total. These do not directly violate security, but do impede the use of a secure protocol. The remaining security-related bugs did not fall into any broader categories.

Our classification of bugs as security-related has some uncertainty, because the bug databases might incorrectly categorize some non-exploitable bugs as security holes. Conversely, some bugs that are not documented as being security-related might be exploitable by a sufficiently clever attacker. In general, any bug in a network protocol implementation is undesirable.

4.4 Results for Extended Rule Sets

In a second set of experiments, we were interested in estimating how easily the rule-based specification could be extended to catch specific bugs we found in the bug databases. Our goal was to study how Pistachio could be used during the development process. In particular, as a programmer finds bugs in their code, good software engineering practice is to write regression tests to catch the bug again if it appears in the future. In the same way, using Pistachio the programmer can write extra "regression rules" to re-run in the future.

We looked for bugs we missed using the core set of rules and added slightly over 10% more rules (9 new rules for LSH and 7 for RCP) to the initial specifications to cover most of the bugs. We found the rules we needed to add were typically for features that were strongly recommended but not required by the specification, because it turned out that violations of these recommendations were considered errors. One example is the recommendation that a proper disconnect message be sent by the SSH server when authentication fails.

Figure 15 shows the effect of the additional rules on the number of the errors detected. In each table we show the specific implementation version, the number of bugs in the database, the total number of warnings from Pistachio and the number of warnings generated from the 10% additional rules. The last two rows in each table in Figure 15 contain the number of false positives caused by the additional rules and the number of false negatives that remain after enriching the specification. We can see that the additional rules account for under 20% of the total number of warnings generated by Pistachio. From the set of new warnings produced by Pistachio after introducing the additional rules, approximately 18% had security implications according to our classification from Section 4.3, mostly related to access control issues and buffer overflows. Roughly half of the remaining false negatives are due to terminating iteration after *max_pass* times, and the other half are due to aspects of the protocol our new rules still did not cover.

For the extended rules, we also measured how often we are able to compute a symbolic fixpoint for loops during our analysis. Recall that if we stop iteration of our

Version	0.1.3	0.2.1	0.2.9	0.9.1	1.0.1	1.1.1	1.2.1	1.3.1	1.4.1	1.5.1	1.5.5	2.0.1
Bugs in database	91	83	69	65	81	80	82	82	51	40	31	13
Warnings from Pistachio	133	143	124	112	106	109	111	100	93	83	40	28
From 10% rules	15	20	14	15	15	16	19	18	15	8	7	3
False pos. from 10%	8	8	6	8	7	5	7	4	6	5	3	2
False neg. after 10%	1	0	1	0	2	0	0	0	1	0	1	0

(a) LSH implementations

Version	0.5.4	0.6.4	0.8.4	1.1.4	1.2.3	1.3.2
Bugs in database	51	46	47	51	28	23
Warnings from Pistachio	84	78	67	77	56	45
From 10% rules	11	9	10	7	8	6
False pos. from 10%	6	5	5	4	2	4
False neg. after 10%	1	1	1	0	1	0

(b) RCP implementations

Figure 15: Analysis Results for Extended Rule Sets

algorithm after *max_pass* times then we could introduce unsoundness, which accounts for approximately 27% of the false positives, as shown in Figure 14. We found that when *max_pass* is set to 75, we find a fixpoint before reaching *max_pass* in 250 out of 271 cases for LSH, and in 153 out of 164 cases for RCP. This suggests that our symbolic fixpoint computation is effective in practice.

5 Related Work

Understanding the safety and robustness of network protocols is recognized as an important research area, and the last decade has witnessed an emergence of many techniques for verifying protocols.

We are aware of only a few systems that, like Pistachio, directly check source code rather than abstract models of protocols. CMC [24] and VeriSoft [15] both model check C source code by running the code dynamically on hardware, and both have been used to check communication protocols. These systems execute the code in a simulated environment in which the model checker controls the execution and interleaving of processes, each of which represents a communication node. As the code runs, the model checker looks for invalid states that violate user-specified assertions, which are similar to the rules in Pistachio. CMC has been successfully used to check an implementation of the AODV routing protocol [24] and the Linux TCP/IP protocol [23].

There are two main drawbacks to these approaches. First, they potentially suffer from the standard state space explosion problem of model checking, because the number of program executions and interleavings is extremely large. This is typical when model checking is used for data dependent properties, and both CMC and VeriSoft use various techniques to limit their search. Second, these tools find errors only if they actually occur during execution, which depends on the number of simulated processes and on what search algorithm is used. Pistachio makes different tradeoffs. Because we start from a set of rules describing the protocol, we need only perform abstract interpretation on a single instance of the protocol rather than simulating multiple communication nodes, which improves performance. The set of rules can be refined over time to find known bugs and make sure that they do not appear again. We search for errors by program source path rather than directly in the dynamic execution space, which means that in the best case we are able to use symbolic information in the dataflow facts to compute fixpoints for loops, though in the worst case we unsafely cut off our search after *max_pass* iterations. Pistachio is also very fast, making it easy to use during the development process. On the other hand, Pistachio's rules cannot enforce the general kinds of temporal properties that model checking can. We believe that ultimately the CMC and VeriSoft approach and the Pistachio approach are complementary, and both provide increased assurance of the safety of a protocol implementation.

Other researchers have proposed static analysis systems that have been applied to protocol source code. MAGIC [7] extracts a finite model from a C program using various abstraction techniques and then verifies the model against the specification of the program. MAGIC has been successfully used to check an implementation of the SSL protocol. The SPIN [18] model checker has been used to trace errors in data communication protocols, concurrent algorithms, and operating systems. It uses a high level language to specify system descriptions but also provides direct support for the use of embedded C code as part of model specifications. However, due to the state space explosion problem, neither SPIN nor MAGIC perform well when verifying data-driven properties of protocols, whereas Pistachio's rules are designed to include data modeling. Feamster and Balakrishnan [14] define a high-level model of the BGP routing protocol by abstracting its configuration. They use this model to build *rcc*, a static analysis tool that detects faults in the router configuration. Naumovich *et al.* [25] propose the FLAVERS tools, which uses dataflow analysis techniques to verify Ada pseudocode for communication protocols. Alur and Wang [2] formulate the problem of verifying a protocol implementation with respect to its standardized documentation as refinement checking. Implementation and specification models are manually extracted from the code and the RFC document and are compared against each other using reachability analysis. The method has been successfully applied to two popular network protocols, PPP and DHCP.

Many systems have been developed for verifying properties of abstract protocol specifications. In these systems the specification is written in a specialized language that usually hides some implementation details. These methods can perform powerful reasoning about protocols, and indeed one of the assumptions behind Pistachio is that the protocols we are checking code against are already well-understood, perhaps using such techniques. The main difficulty of checking abstract protocols is translating RFCs and other standards documents into the formalisms and in picking the right level of abstraction. $Mur\varphi$ is a system for checking protocols in which abstract rules can be extracted from actual C code [20]. The main differences between our approach and the $Mur\varphi$ system lies in how the rules are interpreted: in $Mur\varphi$ the rules are an abstraction of the system and are derived automatically, whereas in Pistachio rules specify the actual properties to be checked in the code. Uppaal [6] models systems (including network protocols) as timed automata, in which transitions between states are guarded by temporal conditions. This type of automata is very useful in checking security protocols that use time challenges and has been used extensively in the literature to that extent [28, 12]. In [12], Uppaal is used to model check the TLS handshake protocol. CTL model checking can also be used to check network protocols. In [9], an extension of the CTL semantics is used to model AODV.

Recently there has been significant research effort on developing static analysis tools for finding bugs in software. We list a few examples: SLAM [3] and BLAST [17] are model checking systems that have been used to find errors in device drivers. MOPS [8] uses model checking to check for security property violations, such as TOCTTOU bugs and improper usage of *setuid*. Metal [13] uses data flow analysis and has been used to find many errors in operating system code. ESP [11] uses data flow analysis and symbolic execution, and has been used to check sequences of I/O operations in gcc. All of these systems have been effective in practice, but do not reason about network protocol implementations, and it is unclear whether they can effectively check the kinds of data-dependent rules used by Pistachio.

Dynamic analysis can also be used to trace program executions, although we have not seen this technique used to check correctness of implementations. Gopalakrishna et al. [16] define an Inlined Automaton Model (IAM) that is flow- and context-sensitive and can be derived from source code using static analysis. The model is then used for online monitoring for intrusion detection.

Another approach to finding bugs in network protocols is online testing. Protocol fuzzers [27] are popular tools that look for vulnerabilities by feeding unexpected and possibly invalid data to a protocol stack. Because fuzzers can find hard-to-anticipate bugs, they can detect vulnerabilities that a Pistachio user might not think to write a rule for. On the other hand, the inherent randomness of fuzzers makes them hard to predict, and sometimes finding even a single bug with fuzzing may take a long time. Pistachio quickly checks for many different bugs based on a specification, and its determinism makes it easier to integrate in the software development process.

Our specification rules are similar to precondition/postcondition semantics usually found in software specification systems or design-by-contract systems like JML [19]. Similar constructs in other verification systems also include BLAST's event specifications [17].

6 Conclusion

We have defined a rule-based method for the specification of network protocols which closely mimics protocol descriptions in RFC or similar documents. We have then shown how static analysis techniques can be employed in checking protocol implementations against the rule-based specification and provided details about our experimental prototype, Pistachio. Our experimental results show that Pistachio is very fast and is able to detect a number of security-related errors in implementations of the SSH2 and RCP protocols, while maintaining low rates of false positives and negatives.

Acknowledgements

This research was supported in part by NSF CCF-0346982 and CCF-0430118. We thank Mike Hicks, David Wagner, Nick Petroni, R. Sekar, and the anonymous reviewers for their helpful comments.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1988.

[2] R. Alur and B.-Y. Wang. Verifying network protocol implementations by symbolic refinement checking. In *Proceedings of International Conference on Computer-Aided Verification*, 2001.

[3] T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of POPL*, 2002.

[4] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, 1969.

[5] P. Baumgartner, A. Fuchs, and C. Tinelli. Darwin: A Theorem Prover for the Model Evolution calculus. In *IJCAR Workshop on Empirically Successful First Order Reasoning*, 2004.

[6] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems. In *Hybrid Systems*, pages 232–243, 1995.

[7] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. In *Proceedings of the 25th International Conference on Software Engineering*, 2003.

[8] H. Chen and D. Wagner. MOPS: An Infrastructure for Examining Security Properties of Software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002.

[9] R. Corin, S. Etalle, P. H. Hartel, and A. Mader. Timed Model Checking of Security Protocols. In *Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering*, 2004.

[10] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[11] M. Das, S. Lerner, and M. Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–68, Berlin, Germany, June 2002.

[12] G. Diaz, F. Cuartero, V. Valero, and F. Pelayo. Automatic Verification of the TLS Handshake Protocol. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, 2004.

[13] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, Oct. 2001.

[14] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *Proceedings of NSDI*, 2005.

[15] P. Godefroid. Model Checking for Programming Languages Using Verisoft. In *Proceedings of POPL*, 1997.

[16] R. Gopalakrishna, E. H. Spafford, and J. Vitek. Efficient intrusion detection using automaton inlining. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2005.

[17] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. *Lecture Notes in Computer Science*, 2648:235–239, January 2003.

[18] G. J. Holzmann. The Model Checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[19] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion*, 2000.

[20] D. Lie, A. Chou, D. Engler, and D. L. Dill. A Simple Method for Extracting Models for Protocol Code. In *Proceedings of the 28th Int'l Symposium on Computer Architecture*, 2001.

[21] A GNU implementation of the Secure Shell protocols. `http://www.lysator.liu.se/~nisse/lsh/`.

[22] N. Möller. lshd leaks fd:s to user shells, Jan. 2006. `http://lists.lysator.liu.se/pipermail/lsh-bugs/2006q1/000467.html`.

[23] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *Proceedings of NSDI*, 2004.

[24] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of OSDI*, 2002.

[25] G. N. Naumovich, L. A. Clarke, and L. J. Osterweil. Verification of Communication Protocols Using Data Flow Analysis. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1996.

[26] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction*, 2002.

[27] P. Oehlert. Violating assumptions with fuzzing. *IEEE Security & Privacy Magazine*, 3:58–62, 2005.

[28] S. Yang and J. S. Baras. Modeling Vulnerabilities of Ad Hoc Routing Protocols. In *Proceedings of the 1st ACM workshop on Security of Ad Hoc and Sensor Networks*, 2003.

Evaluating SFI for a CISC Architecture

Stephen McCamant
Massachusetts Institute of Technology
Computer Science and AI Lab
Cambridge, MA 02139
smcc@csail.mit.edu

Greg Morrisett
Harvard University
Division of Engineering and Applied Sciences
Cambridge, MA 02138
greg@eecs.harvard.edu

Abstract

Executing untrusted code while preserving security requires that the code be prevented from modifying memory or executing instructions except as explicitly allowed. Software-based fault isolation (SFI) or "sandboxing" enforces such a policy by rewriting the untrusted code at the instruction level. However, the original sandboxing technique of Wahbe et al. is applicable only to RISC architectures, and most other previous work is either insecure, or has been not described in enough detail to give confidence in its security properties. We present a new sandboxing technique that can be applied to a CISC architecture like the IA-32, and whose application can be checked at load-time to minimize the TCB. We describe an implementation which provides a robust security guarantee and has low runtime overheads (an average of 21% on the SPECint2000 benchmarks). We evaluate the utility of the technique by applying it to untrusted decompression modules in an archive tool, and its safety by constructing a machine-checked proof that any program approved by the verification algorithm will respect the desired safety property.

1 Introduction

Secure systems often need to execute a code module while constraining its actions with a security policy. The code might come directly from a malicious author, or it might have bugs that allow it to be subverted by maliciously chosen inputs. The system designer chooses a set of legal interfaces for interaction with the code, and the challenge is to ensure that the code's interaction with the rest of the system is limited to those interfaces. Software-based fault isolation (SFI) implements such isolation via instruction rewriting, but previous research left the practicality of the technique uncertain. The original SFI technique works only for RISC architectures, and much followup research has neglected key security issues. By contrast, we find that SFI can be implemented for the x86 with runtime overheads that are acceptable for many applications, and that the technique's security can be demonstrated with a rigorous machine-checked proof.

The most common technique for isolating untrusted code is the use of hardware memory protection in the form of an operating system process. Code in one process is restricted to accessing memory only in its address space, and its interaction with the rest of a system is limited to a predefined system call interface. The enforcement of these restrictions is robust and has a low overhead because of the use of dedicated hardware mechanisms such as TLBs; few restrictions are placed on what the untrusted code can do. A disadvantage of hardware protection, however, is that interaction across a process boundary (i.e., via system calls) is coarse-grained and relatively expensive. Because of this inefficiency and inconvenience, it is still most common for even large applications, servers, and operating system kernels to be constructed to run in a single address space.

A very different technique is to require that the untrusted code be written in a type-safe language such as Java. The language's type discipline limits the memory usage and control flow of the code to well-behaved patterns, making fine-grained sharing of data between trusted and untrusted components relatively easy. However, type systems have some limitations as a security mechanism. First, unlike operating systems, which are generally language independent, type system approaches are often designed for a single language, and can be hard to apply to at all to unsafe languages such as C and C++. Second, conventional type systems describe high-level program actions like method calls and field accesses. It is much more difficult to use a type system to constrain code at the same level of abstraction as individual machine instructions; but since it is the actual instructions that will be executed, only a safety property in terms of them would be really convincing.

This paper investigates a code isolation technique that

```
void f(int arg,  int arg2,
       int arg3, int arg4) {
    return;
}
void poke(int *loc, int val) {
    int local;
    unsigned diff = &local - loc - 2;
    for (diff /= 4; diff; diff--)
        alloca(16);
    f(val, val, val, val);
}
```

Figure 1: Example attack against SFI systems which depend on the compiler's management of the stack for safety. The function `poke` modifies the stack pointer by repeatedly allocating unused buffers with `alloca` until it points near an arbitrary target location `loc`, which is then overwritten by one of the arguments to `f`. MiSFIT [25] and Erlingsson's x86 SASI tool [10] allow this unsafe write because they incorrectly assume that the stack pointer always points to the legal data region.

lies between the approaches mentioned above, one that enforces a security policy similar to an operating system, but with ahead-of-time code verification more like a type system. This effect is achieved by rewriting the machine instructions of code after compilation to directly enforce limits on memory writes and control flow. This class of techniques is known as "software-based fault isolation" (SFI for short) or "sandboxing" [27]. Previous SFI techniques were applicable only to RISC architectures, or their treatment of key security issues was incomplete, faulty, or never described publicly. For instance, several previous systems [25, 10] depended for their safety on an assumption that a C compiler would manage the stack pointer to keep the untrusted code's stack separate from the rest of memory. As shown in the example of Figure 1, this trust is misplaced, not just because compilers are large and may contain bugs, but because the safety guarantees they make are loosely specified and contain exceptions. (Concurrently with the research described here, some other researchers have also developed SFI-like systems that include more rigorous security analyses; see Section 10 for discussion.)

Many systems programming applications can benefit from a code isolation mechanism that is efficient, robust, and easily applicable to existing code. A useful technique to improve the reliability of operating systems is to isolate device drivers so that their failures (which may include arbitrary memory writes) do not corrupt the rest of a kernel. The Nooks system [26] achieves such isolation with hardware mechanisms that are robust, but impose a high overhead when many short cross-boundary calls are made; SFI could provide similar protection without high per-call overheads. To reduce the damage caused by attacks on network servers, they should be designed to minimize the amount of code that requires high (e.g., root) privileges; but retrofitting such a design on an existing server is difficult. Dividing servers along these lines by using separate OS-level processes [23, 14] is effective at preventing vulnerabilities, but is far from trivial because of the need to serialize data for transport and prevent an untrusted process from making damaging system calls. SFI could make such separation easier by automatically preventing system calls and making memory sharing more transparent. Section 8 discusses VXA [11], an architecture that ensures compressed archives will be readable in the future by embedding an appropriate decompressor directly in an archive. Applying our SFI tool to VXA we see that it very easily obtains a strong security guarantee, without imposing prohibitive runtime overheads. Note that all of these examples include large existing code bases written in C or C++, which would be impractical to rewrite in a new language; the language neutrality of SFI techniques is key to their applicability.

This paper:

- Describes a novel SFI technique directly applicable to CISC architectures like the Intel IA-32 (x86), as well as two optimizations not present in previous systems (Sections 3 and 4).
- Explains how using separate verification, the security of the technique depends on a minimal trusted base (on the order of a thousand lines of code), rather than on tools consisting of hundreds of thousands of lines (Section 5).
- Analyzes in detail the performance of an implementation on the standard SPECint2000 benchmarks (Section 7).
- Evaluates the implementation as part of a system to safely execute embedded decompression modules (Section 8).
- Gives a machine-checked proof of the soundness of the technique (specifically, of the independent safety verifier) to provide further evidence that it is simple and trustworthy (Section 9).

We refer to our implementation as the Prototype IA-32 Transformation Tool for Software-based Fault Isolation Enabling Load-time Determinations (of safety), or PittSFIeld[1]. Our implementation is publicly available (the version described here is 0.4), as are the formal model and lemmas used in the machine-checked proof. They can be downloaded from the project web site at `http://pag.csail.mit.edu/~smcc/projects/pittsfield/`.

[1]Pittsfield, Massachusetts, population 45,793, is the seat of Berkshire county and a leading center of plastics manufacturing. Our appropriation of its name, however, was motivated only by spelling.

2 Classic SFI

The basic task for any SFI implementation is to prevent certain potentially unsafe instructions (such as memory writes) from being executed with improper arguments (such as an effective address outside an allowed data area). The key challenges are to perform these checks efficiently, and in such a way that they cannot be bypassed by maliciously designed jumps in the input code. The first approach to solve these challenges was the original SFI technique (called "sandboxing") of Wahbe, Lucco, Anderson, and Graham [27]. (The basic idea of rewriting instructions for safety had been suggested earlier, notably by Deutsch and Grant [7], but their system applied to code fragments more limited than general programs.)

In order to efficiently isolate pointers to dedicated code and data regions, Wahbe et al. suggest choosing memory regions whose size is a power of two, and whose starting location is aligned to that same power. For instance, we might choose a data region starting at the address `0xda000000` and extending 16 megabytes to `0xdaffffff`. With such a choice, an address can be efficiently checked to point inside the region by bitwise operations. In this case, we could check whether the bitwise AND of an address and the constant `0xff000000` was equal to `0xda000000`. We'll use the term *tag* to refer to the portion of the address that's the same for every address in a region, such as `0xda` above.

The second challenge, assuring that checks cannot be bypassed, is more subtle. Naively, one might insert a checking instruction sequence directly before a potentially unsafe operation; then a sequential execution couldn't reach the dangerous operation without passing through the check. However, it isn't practical to restrict code to execute sequentially: realistic code requires jump and branch instructions, and with them comes the danger that execution will jump directly to an dangerous instruction, bypassing a check. Direct branches, ones in which the target of the branch is specified directly in the instruction, are not problematic: a tool can easily check their destinations before execution. The crux of the problem is indirect jump instructions, ones where the target address comes from a register at runtime. They are required by procedure returns, `switch` statements, function pointers, and object dispatch tables, among other language features (Deutsch and Grant's system did not allow them). Indirect jumps must also be checked to see that their target address is in the allowed code region, but how can we also exclude the addresses of unsafe instructions, while allowing safe instruction addresses?

The key contribution of Wahbe et al. was to show that by directing all unsafe operations through a dedicated register, a jump to any instruction in the code region could be safe. For instance, suppose we dedicate the register `%rs` for writes to the data area introduced above. Then we maintain that throughout the code's execution, the value in `%rs` always contains a value whose high bits are `0xda`. Code can only be allowed to store an arbitrary value into `%rs` if it immediately guarantees that the stored value really is appropriate. If we know that this invariant holds whenever the code jumps, we can see that even if the code jumps directly to an instruction that stores to the address in `%rs`, all that will occur is a write to the data region, which is safe (allowed by the security policy). Of course, there is no reason why a correct program *would* jump directly to an unsafe store instruction; the technique is needed for incorrect or maliciously designed programs.

Wahbe et al. implemented their technique for two RISC architectures, the MIPS and the Alpha. Because memory reads are more common than writes and less dangerous, their implementation only checked stores and not loads, a tradeoff that has also been made in most subsequent work, including ours. (In the experiments in [25], adding protection for out-of-bounds reads often more than doubled overhead compared to checking only writes and jumps.) Because separate dedicated registers are required for the code and data regions, and because constants used in the sandboxing operation also need to be stored in registers, a total of 5 registers are required; out of a total of 32, the performance cost of their loss was negligible. Wahbe et al. evaluated their implementation by using it to isolate faults in an extension to a database server. While fault isolation decreases the performance of the extension itself, the total effect is small, significantly less than the overhead of running the extension run in a separate process, because communication between the extension and the main server becomes inexpensive.

3 CISC architectures

The approach of Wahbe et al. is not immediately applicable to CISC architectures like the Intel IA-32 (i386 or "x86"), which feature variable-length instructions. (The IA-32's smaller number of registers also makes dedicating several registers undesirable, though its 32-bit immediates mean that only 2 would be needed.) Implicit in the previous discussion of Wahbe et al.'s technique was that jumps were restricted to a single stream of instructions (each 4-byte aligned, in a typical RISC architecture). By contrast, the x86 has variable-length instructions that might start at any byte. Typically code has a single stream of intended instructions, each following directly after the last, but by starting at a byte in the middle of an intended instruction, the processor can read an alternate stream of instructions, generally nonsensical. If code were allowed to jump to any byte

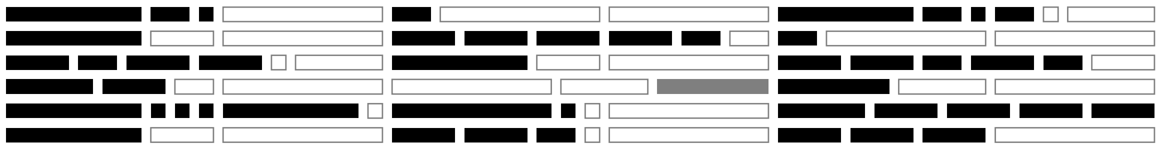


Figure 2: Illustration of the instruction alignment enforced by our technique. Black filled rectangles represent instructions of various lengths present in the original program. Gray outline rectangles represent added no-op instructions. Instructions are not packed as tightly as possible into chunks because jump targets must be aligned, and because the rewriter cannot always predict the length of an instruction. Call instructions (gray filled box) go at the end of chunks, so that the addresses following them are aligned.

offset, the SFI implementation would need to check the safety of all of these alternate instruction streams; but this would be infeasible. The identity of the hidden instructions is a seemingly random function of the precise encodings of the intended ones (including for instance the eventual absolute addresses of forward jump targets), and most modifications to hidden instructions would garble the real ones.

To avoid this problem, our PittSFIeld tool artificially enforces its own alignment constraints on the x86 architecture. Conceptually, we divide memory into segments we call *chunks* whose size and location is a power of two, say 16, bytes. PittSFIeld inserts no-op instructions as padding so that no instruction crosses a chunk boundary; every 16-byte aligned address holds a valid instruction. Instructions that are targets of jumps are put at the beginning of chunks; `call` instructions go at the ends of chunks, because the instructions after them are the targets of returns. This alignment is illustrated schematically in Figure 2. Furthermore, jump instructions are checked so that their target addresses always have their low 4 bits zero. This transformation means that each chunk is an atomic unit of execution with respect to incoming jumps: it is impossible to execute the second instruction of a chunk without executing the first. To ensure that an otherwise unsafe operation and the check of its operand cannot be separated, PittSFIeld additionally enforces that such pairs of instructions do not cross chunk boundaries, making them atomic. Thus, our technique does not need dedicated registers as in classic SFI. A scratch register is still required to hold the effective address of an operation while it is being checked, but it isn't required that the same register be used consistently, or that other uses of the register be prohibited. (For reasons of implementation simplicity, though, our current system consistently uses `%ebx`.)

4 Optimizations

The basic technique described in Section 3 ensures the memory and control-flow safety properties we desire, but as described it imposes a large performance penalty. This section describes five optimizations that reduce the overhead of the rewriting process, at the expense of making it somewhat more complex. The first three optimizations were described by Wahbe et al., and are well known; the last two have, as far as we know, not previously been applied to SFI implementations.

Special registers. The register `%ebp` (the 'frame pointer' or 'base pointer') is often used to access local variables stored on the stack, part of the data region. Since `%ebp` is generally set only at the start of a function but then used repeatedly thereafter, checking its value at each use is inefficient. A better strategy is to make sure that `%ebp`'s value is a safe data pointer everywhere by checking its value after each modification. This policy treats `%ebp` like the reserved registers of Wahbe et al., but since `%ebp` is already reserved by the ABI for this purpose, the number of available general-purpose registers is not decreased.

Guard regions. The technique described in the previous paragraph for optimizing the use of `%ebp` would be effective if `%ebp` were only dereferenced directly, but in fact `%ebp` is often used with a small constant offset to access the variables in a function's stack frame. Usually, if `%ebp` is in the data region, then so is `%ebp + 10`, but this would not be the case if `%ebp` were already near the end of the data region. To handle this case efficiently, we follow Wahbe et al. in using *guard regions*, areas in the address space directly before and after the data region that are also safe for the sandboxed code to attempt to write to.

If we further assume that accesses to the guard region can be efficiently trapped (such as by leaving them unmapped in the page table), we can optimize the use of the stack pointer `%esp` in a similar way. The stack pointer is similar to `%ebp` in that it generally points to the stack and is accessed at small offsets, but unlike the frame pointer, it is frequently modified as items are pushed onto and popped off the stack. Even if each individual change is small, each must be checked to make sure that it isn't the change that pushes `%esp` past the end of the allowable region. However, if attempts to access the guard regions are trapped, every use of `%esp` can also serve as a check of the new value. One important point is that we must be careful of modifications of `%esp` that do not also use it.

The danger of a sequence of small modifications is illustrated in the example of Figure 1: each call to `alloca` decrements `%esp` by a small amount but does not use it to read or write. Our system prevents this attack by requiring a modified `%esp` to be checked before a jump.

Ensure, don't check. A final optimization that was included in the work of Wahbe et al. has to do with the basic philosophy of the safety policy that the rewriting enforces. Clearly, the untrusted code should not be able to perform any action that is unsafe; but what should happen when the untrusted code attempts an unsafe action? The most natural choice would be to terminate the untrusted code with an error report. Another possibility, however, would be to simply require that when an unsafe action is attempted, some action consistent with the security policy occurs instead. For example, instead of a jump to a forbidden area causing an exception, it might instead cause a jump to some arbitrary other location in the code region. The latter policy can be more efficient because no branch is required: the code simply sets the bits of the address appropriately and uses it. If the address was originally illegal, it will 'wrap around' to some legal, though likely not meaningful, location.

There are certainly applications (such as debugging) where such arbitrary behavior would be unhelpful. However, it is reasonable to optimize a security mechanism for the convenience of legitimate code, rather than of illegal code. Attempted jumps to an illegal address should not be expected to occur frequently in practice: it is the responsibility of the code producer (and her compiler), not the code user, to avoid them. Our rewriting tool supports both modes of operation, but we here follow Wahbe et al.in concentrating on the more efficient ensure-only mode, which we consider more realistic. Experiments described in a previous report [18] show that the checking mode introduces an average of 12% further overhead over the ensure-only mode on some realistic examples.

One-instruction address operations. For an arbitrarily chosen code or data region, the sandboxing instruction must check (or, according to the optimization above, ensure) that certain bits of an address are set, and others are clear. This requires two instructions: an AND instruction to turn some bits off and an OR instruction set others. By further restricting the locations of the sandbox regions, however, the number of instructions can be reduced to one. We choose the code and data regions so that their tags have only a single bit set, and then reserve from use the region of the same size starting at address 0, which we call the *zero-tag region* (because it corresponds to a tag of 0). With this change, bits in the address only need to be cleared, and not also set.

PittSFIeld by default uses code and data regions of 16MB each, starting at the addresses `0x10000000` and `0x20000000` respectively. The code sequence to ensure that an address in `%ebx` is legal for the data region is:[2]

```
    and    $0x20ffffff, %ebx
```

This instruction turns off all of the bits in the tag except possibly the third from the top, so the address will be either in the data region or the zero-tag region. On examples such as the set of larger programs appearing in a previous report [18], disabling this optimization increases PittSFIeld's overhead over normal execution by about 10%.

Efficient returns. A final optimization helps PittSFIeld take advantage of the predictive features of modern processors. Indirect jumps are potentially expensive for processors if their targets cannot be accurately predicted. For general indirect jumps, processors typically keep a cache, called a 'branch target buffer', of the most recent target for a jump instruction. A particularly common kind of indirect jump is a procedure return, which on the x86 reads a return address from the stack. A naive implementation would treat a return as a pop followed by a standard indirect jump; for instance, an early version of PittSFIeld translated a `ret` instruction into (in this example and the next, the final two instructions must be in a single chunk):

```
    popl   %ebx
    and    $0x10fffff0, %ebx
    jmp    *%ebx
```

However, if a procedure is called from multiple locations, the single buffer slot will not be effective at predicting the return address, and performance will suffer. In order to deal more efficiently with returns, modern x86 processors keep a shadow stack of return addresses in a separate cache, and use this to predict the destinations of returns. To allow the processor to use this cache, we would like PittSFIeld to return from procedures using a real `ret` instruction. Thus PittSFIeld modifies the return address and writes it back to the stack before using a regular `ret`. In fact, this can be done without a scratch register:

```
    and    $0x10fffff0, (%esp)
    ret
```

On a worst case example, like a recursive implementation of the Fibonacci function, this optimization makes an enormous difference, reducing 95% overhead to 40%. In more realistic examples, the difference is smaller but still significant; for the SPECint2000 benchmarks discussed in Section 7, disabling this optimization increases

[2]Assembly language examples use the GAS, or 'AT&T', syntax standard on Unix-like x86-based systems, which puts the destination last.

the average overhead from 21% to 27%, and almost doubles the overhead for one program, 255.vortex.

With the exception of this optimization, the rest of the PittSFIeld system can maintain its security policy even if arbitrary changes to the data region occur between instructions, because instructions always move addresses to registers before checking them. However, the `ret` instruction unavoidably uses the stack, so this optimization is applicable under the more limited attack model in which untrusted data changes come from a single untrusted thread. The optimization should not be used if multiple threads run in the same data sandbox, or if other untrusted memory changes (such as memory-mapped I/O) might occur in parallel.

5 Verification

The intended use of PittSFIeld is that the compilation and the rewriting of the code are performed by the untrusted code producer, and the safety policy is enforced by a separate verification tool. This architecture is familiar to users of Java: the code producer writes source code and compiles it to byte code using the compiler of her choice, but before the code user executes an applet he checks it using a separate byte code verifier. (One difference from Java is that once checked, our code is executed more or less directly; there is no trusted interpreter as complex as a Java just-in-time compiler.) The importance of having a small, trusted verifier is also stressed in work on proof-carrying code [20]. Though the advantages of this architecture are well known, they have been neglected by some previous SFI implementations, leading to predictable problems with usability and security (see Section 10.1).

Responsibility for ensuring the safety of execution in the PittSFIeld system lies with a verifier which examines the rewritten code just prior to execution, conservatively checking properties which, if true, ensure that execution of the code will not violate the system's security policy. In a more complex system, one could imagine the rewriting process supplying hints describing why the rewritten code satisfies the security policy (like the proof in a proof-carrying code system), but PittSFIeld's policies are simple enough that this is not necessary. In particular, the verifier does not require debugging or symbol-table information; the verifier must disassemble the rewritten code, but the rewriter ensures that the disassembly can be performed in a single pass without respect to function boundaries. The role of the verifier is to prove that the rewritten code is safe, so its design is best thought of as automating such a proof. Section 9 will describe in more detail how that intuitive proof can be formalized.

To understand how the verification works, it is helpful to borrow concepts from program analysis, and think of it as a conservative static analysis. The verifier checks a property of the program's execution, roughly that it never jumps outside its code region or writes outside its data region. In general, this property is impossible to decide, but it is tractable if we are willing to accept one-sided error: we do not mind if the verifier fails to recognize that some programs have the safety property, as long as whenever it concludes that one does, it is correct. If the original program was correct, it already had this safety property; the rewriting simply makes the property manifest, so that the verifier can easily check it.

The verification process essentially computes, for each position in the rewritten instruction stream, a conservative property describing the contents of the processor's registers at any time when execution might reach that point. For instance, directly after an appropriate `and` instruction not at a chunk boundary, we might know that the contents of the target register are appropriately sandboxed for use in accessing the data region. The major part of the safety proof is to show that these properties are sound for any possible execution; it is then easy to see that if the properties always hold, no unsafe executions will be possible. An important aspect of the soundness is that it is inductive over the steps in the execution of the rewritten code: for instance, it is important that none of the instructions in the code region change during execution, as new instructions would not necessarily match the static properties. We can be confident of this only because in previous execution up to a given point, we can assume we were successful in preventing writes outside the data section. In program verification terminology, the soundness property is an invariant that the verifier checks as being preserved by each instruction step.

6 Prototype implementation

To test the practicality of our approach, we have constructed a prototype implementation, named PittSFIeld. PittSFIeld instantiates a simple version of the technique, incorporating only the most important optimizations. However, PittSFIeld was designed to address some important practical considerations for a real tool, such as the separate verification model and scalability to large and complex programs. In particular, PittSFIeld makes no fundamental compromises with respect to the rigorous security guarantees that the technique offers. The performance of code rewritten by PittSFIeld (described in the next section) should also give a reasonable upper bound on the overhead of this general approach, one which could be somewhat improved by further optimization. (However, other aspects of the prototype are not representative of a practical implementation: for instance, the rewriter is unrealistically slow.)

The rewriting performed by PittSFIeld is a version of the techniques described in Sections 3 and 4, chosen to be easy to perform. The register `%ebx` is reserved (using the `--fixed-ebx` flag to GCC), and used to hold the sandboxed address for accesses to both the data and code regions. The effective address of an unsafe operation is computed in `%ebx` using a `lea` instruction. The value in `%ebx` is required to be checked or sandboxed directly before each data write or indirect code jump (reads are unrestricted). Both direct and indirect jumps are constrained to chunk-aligned targets. Guard regions are 64k bytes in size: `%ebp` and `%esp` are treated as usually-sandboxed. Accesses are allowed at an offset of up to 64k from `%ebp`, and of up to 255 bytes from `%esp`; `%esp` is also allowed to be modified up to 255 times, by as much as 255 bytes each time, between checks. Both `%ebp` and `%esp` must be restored to safe values before a jump. A safe value in `%esp` may be copied to `%ebp` or vice-versa without a check. Chunks are padded using standard no-op instructions of lengths 1, 2, 3, 4, 6, and 7 bytes, to a size of 16 or 32 bytes.

Because it operates on assembly code, our prototype rewriting tool is intended to be used by a code producer. A system that instead operates on off-the-shelf binaries without the code producer's cooperation is often described as a goal of SFI research, but has rarely been achieved in practice. The key difficulty is that binaries do not contain enough information to adjust jumps when instructions are added: for instance, it may not be possible to distinguish between an address referring to an instruction and an integer with the same numeric value. A more feasible approach is to operate on binaries supplemented with additional relocation information, such as the debugging information used by the Vulcan library in [1], or the SELF extension to ELF proposed in [8].

Both the rewriting and the verification in PittSFIeld are performed as single top-to-bottom passes, essentially as finite-state machines. While this prohibits some optimizations (for instance, labels that are targets only of direct jumps need not necessarily be aligned), it allows PittSFIeld to rewrite very large programs, and guarantees that the verification's running time will be linear. (A verification technique with bad worst-case performance can allow a denial-of-service attack [12]).

The rewriting phase of PittSFIeld is implemented as a text processing tool, of about 720 lines of code, operating on input to the GNU assembler `gas`. In most cases, alignment is achieved using the `.p2align` directive to the assembler, which computes the correct number of no-ops to add; the rewriter uses a conservative estimate of instruction length to decide when to emit a `.p2align`. The rewriter adds no-ops itself for aligning call instructions, because they need to go at the end rather than the beginning of a chunk. The rewriter notices instructions that are likely to be used for their effect on the processor status flags (e.g., comparisons), and saves and restores the flags register around sandboxing operations when the flags are deemed live. However, such save and restore operations can be costly on modern architectures, so to prevent GCC from moving comparisons away from their corresponding branches, we disable instruction scheduling with the `-fno-schedule-insns2` option when compiling for PittSFIeld. An example of the rewriter's operation on a small function is shown in Figure 3.

We have implemented two prototypes for the verification phase of PittSFIeld, which implement the same algorithm. Because they use a single disassembly pass, the verifiers enforce alignment by checking that an instruction in the single stream must appear at each chunk starting address. The verifiers currently verify only the style of rewriting in which pointers are modified, and not the style in which they are checked and execution halted if they are incorrect. As mentioned in Section 5, the verifiers are essentially finite-state: at each code location, they keep track of variations from the standard safety invariant, checking them and then updating their knowledge for each instruction. Operations that 'strengthen' the invariant (for instance, sandboxing a pointer value in `%ebx`) expire after one instruction or at a chunk boundary, whichever comes first. Operations that 'weaken' the invariant (for instance, loading a new value into `%ebp`) persist until corrected, and must not reach a jump.

Our first verifier is implemented using the same text-processing framework as the rewriter: it is a filter that parses the output of the disassembler from the GNU "binutils" package (the program named `objdump`), and represents about 500 lines of code. Our second verifier is implemented directly in the program that loads and executes sandboxed code objects, using a pre-existing disassembly library; this allows for a better assessment of the performance overheads of verification. Though it does not yet check the complete safety policy, the second verifier is complete enough to give rough performance estimates: for instance, it can verify the 2.7MB of rewritten code for GCC, the largest of the programs from Section 7, in about half a second. Both of our verifiers are much smaller than the disassemblers they use, so the total amount of trusted code could be reduced by disassembling only to the extent needed for verification, but using existing disassemblers takes advantage of other users' testing. Performing more targeted disassembly in this way would also be a way to further improve performance. PittSFIeld supports a large subset of the x86 32-bit protected mode instruction set, but supervisor mode instructions, string instructions, and multimedia SIMD (e.g. MMX, SSE) instructions are not supported; the verifier will reject any program containing an instruction it does not recognize.

```
f:  push    %ebp                   f:   push   %ebp
    mov     %esp, %ebp                  mov    %esp, %ebp
    mov     8(%ebp), %edx               mov    8(%ebp), %edx
    mov     48(%edx), %eax              mov    48(%edx), %eax
    lea     1(%eax), %ecx               lea    1(%eax), %ecx
                                        lea    0(%esi), %esi
                                   ------------------------------
                                        lea    48(%edx), %ebx
                                        lea    0(%esi), %esi
                                        lea    0(%edi), %edi
                                   ------------------------------
                                        and    $0x20ffffff, %ebx
    mov     %ecx, 48(%edx)              mov    %ecx, (%ebx)
    pop     %ebp                        pop    %ebp
                                        lea    0(%esi), %esi
                                   ------------------------------
                                        and    $0x20ffffff, %ebp
                                        andl   $0x10fffff0, (%esp)
    ret                                 ret
```

Figure 3: Before and after example of code transformation. `f` is a function that takes an array of integers, increments the 12th, and returns (in `%eax`) the value before the increment. The assembly code on the left is produced by GCC; that on the right shows the results of the PittSFIeld rewriter after assembly. Rules separate the chunks, and no-op instructions are shown in gray. (Though they look the same here, the first three no-ops use different instruction encodings so as to take 4, 6, and 7 bytes respectively).

7 Performance results

To asses the time and space overheads imposed by our technique, we used our PittSFIeld tool to run stand-alone applications in fault-isolated environments. The programs were not chosen as code one might particularly want to run from an untrusted source, merely as computation-intensive benchmarks. The 'untrusted' code in each case consisted of the application itself, and some simple standard library routines. More complex library routines and system calls were treated as 'trusted,' and accessed via special stubs allowing controlled access out of the sandbox. In a realistic application, these stubs would include checks of their arguments to enforce desired security policies. In our prototype, the trusted loading application and stub trusted calls consisted of approximately 800 lines of C code, including blank lines and comments. A previous technical report [18] gives results for an older version of PittSFIeld run on a set of microbenchmarks, and some larger applications. For better comparison with other work, we here concentrate on a standard set of compute-intensive programs, the integer benchmarks from the SPEC CPU2000 suite.

The SPECint2000 suite consists of 12 programs and reference inputs intended to test the performance of CPUs, compilers, and memory subsystems. One of the programs is written in C++, and the rest in C. In our tests, we compiled the programs with GCC or G++ version 3.3.5 at the `-O3` optimization level. The test system was a 3.06GHz Pentium 4 "Northwood", with 512KB of cache and 2GB of main memory, running Debian Linux 3.1 with kernel version 2.4.28 and C library version 2.3.2. We changed the layout of the code and data sandbox areas to allow a larger data area. Each test was run five times with the reference inputs using the standard SPEC scripts; we discarded the slowest and fastest runtimes and took the average of the remaining three.

In order to measure the effect on performance of different aspects of PittSFIeld's rewriting, we ran the programs using a number of treatments, representing increasing subsets of the transformation the real tool performs. Figure 4 shows the increase in runtime overhead as each transformation is enabled, from bottom to top. The base treatment uses PittSFIeld's program loader, but compiles the programs with normal optimization and uses none of the features of the rewriter. The measurements of Figure 4 are all measured as percentage overhead relative to the base treatment. The first (bottom) set of bars in Figure 4 represents disabling instruction scheduling with an option to GCC. Disabling this optimization has a small performance penalty, but avoids higher overheads later by reducing the need to save and restore the EFLAGS register as discussed in Section 6. The next set of bars represents the effect of directing GCC to avoid using the `%ebx` register in its generated code, reducing the number of general purpose registers from 6 to 5; PittSFIeld requires `%ebx` to be available to hold the effective address of indirect writes and jumps. The next treatment, labelled "padding", reflects the basic cost of requiring chunk alignment: the rewriter adds enough no-op instructions so that no instruction crosses a 16-byte boundary, and every jump target is 16-byte aligned. The next set of bars, labelled "NOP sandboxing", attempts to measure all of the additional overheads related to PittSFIeld's code size increase, beyond those measured in "padding". To achieve this, this treatment adds just as many bytes of new instructions as PittSFIeld normally would, but makes all of them no-ops: this includes both sandboxing instructions, and additional padding required for the new in-

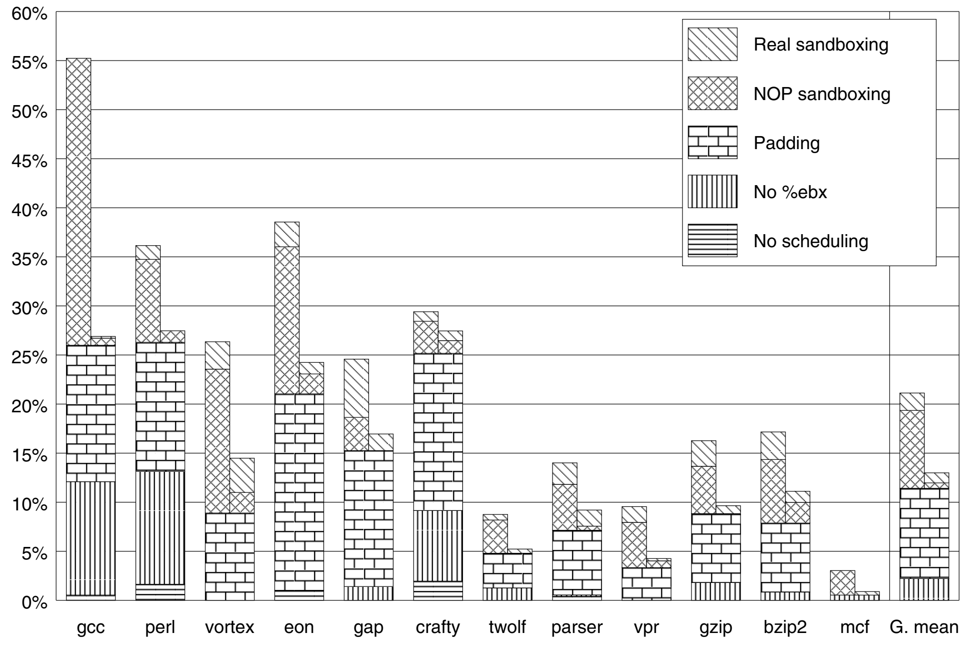


Figure 4: Runtime overheads of PittSFIeld for the SPECint2000 benchmarks, by source. The left half of each bar represents overhead when both jumps and memory writes are protected; the right half shows the overhead of protecting jumps only. The programs are listed in decreasing order of binary size. See the body of the text, Section 7, for details on the meaning of each type of overhead.

structions and to keep some instruction pairs in the same chunk. Finally, the last set of bars represents the complete PittSFIeld transformation; exactly the same number of instructions as "NOP sandboxing", but with AND instructions instead of no-ops as appropriate. For the last two treatments, we also considered another subset of PittSFIeld's rewriting: the left half of each bar shows the overhead when PittSFIeld is used to protect both writes to memory and indirect jumps; the right half shows the overhead for protecting jumps only. For some combinations of programs and treatments, we actually measured a small performance improvement relative to the previous treatment, either because of the inaccuracy of our runtime measurement or because of unpredictable performance effects such as instruction cache conflicts. In these cases the corresponding bars are omitted from the figure.

The SPECint2000 results shown in Figure 4 make clear which of the sources of PittSFIeld's overhead are most significant. Disabling instruction scheduling has little to no effect at this scale, and the sandboxing instructions themselves, bitwise operations on registers, are almost as cheap as no-ops. The effect of reducing the number of available registers varies greatly between programs, but is never the most important overhead. The largest source of overhead is unfortunately the one most fundamental to the technique, the increase in the number of instructions. Added no-op instructions cause two kinds of overhead: first, they take time to execute themselves, and second, they use cache space that would otherwise be used by useful instructions. The relative importance of these two effects can be estimated by comparing the size of the "padding" overhead across programs. Though the proportion of padding instructions can be expected to vary slightly among programs (for instance, being smaller in programs with larger basic blocks), the variation in padding overheads is larger that could be explained by this effect, so the remaining variation must be explained differences in instruction cache pressure. For instance, the padding overhead is larger for large programs than for small ones. The very low overheads for `mcf` likely have two causes: first, it is the smallest of the benchmarks, so instruction cache pressures affect it the least; second, it makes many random accesses to a large data structure, so its runtime depends more on main memory latency than anything happening on the CPU. The final column of Figure 4 shows the average overhead of the technique over all the programs (a geometric mean). This is approximately 21% for memory and jump protection, and 13% for jump protection only.

Figure 5 show how PittSFIeld's transformation affects the size of the code. The row labelled "Size" shows the size of a binary rewritten by PittSFIeld, in bytes (K = 2^{10}, M = 2^{20}). This size includes the program and the stub li-

Program	gcc	perl	vortex	eon	gap	crafty	twolf	parser	vpr	gzip	bzip2	mcf
Size	2.7M	1.2M	1010K	923K	853K	408K	390K	276K	267K	109K	108K	50K
Ratio	1.84	1.96	1.63	1.72	1.84	1.62	1.80	1.92	1.67	1.65	1.63	1.74
Compressed	1.05	1.07	0.98	1.05	1.05	1.06	1.08	1.06	1.07	1.10	1.09	1.13

Figure 5: PittSFIeld space overheads for the SPECint2000 benchmarks. “Size” is the size of the PittSFIeld-rewritten binary. “Ratio” is the ratio of the size of the rewritten binary to the size of a binary generated without rewriting. “Compressed” is like “Ratio”, except with both binaries compressed with `bzip2` before comparing their sizes.

brary, both rewritten by PittSFIeld in its default mode (16-byte chunks, both memory and jump protection). The row “Ratio” shows the ratio of the size of an object file processed by PittSFIeld to that of an unmodified program. The row “Compressed” is analogous, except that both the transformed and original object files were first compressed with `bzip2`. Which of these measurements is relevant depends on the application. The most important effect of PittSFIeld’s size increase in most applications is likely its effect on performance, discussed in the previous paragraph. Uncompressed size is relevant for memory usage, for instance on smaller devices. Compressed size is more relevant, for instance, to the cost of storing and distributing software; the compressed ratios are smaller because the added instructions tend to be repetitive.

8 Application case study

To assess the usability of PittSFIeld for a realistic application, this section investigates using PittSFIeld as the isolation mechanism for VXA, a system in which compressed archives contain their own decompressors [11]. A major challenge to our ability to preserve digital information for the future is the proliferation of incompatible file formats. Compression formats are a particular problem: for instance while uncompressed audio formats have been stable since the early 1990s, four major new formats for compressed audio have become popular since 2000. By comparison, the basic IA-32 architecture retains backwards compatibility with software written for the first 386es in 1985. To exploit these relative rates of change, the VXA system introduces an archive file format and tools called vxZIP and vxUnZIP that extend the well-known ZIP format by including decompressors in a standardized IA-32 binary format inside compressed archives. IA-32 was chosen as the standard decompressor format so that vxZIP and vxUnZIP can be used with low overhead today.

Clearly a key challenge for vxUnZIP is to run the supplied decompressor without allowing a malicious decompressor to compromise security. At the same time, it is desirable for the decompressors to run in the same process as the rest of vxUnZIP. Compared to using a separate OS-level process for isolation, running in a single process avoids performance overheads associated with process creation and copying data between processes, but the most important advantage is the ease of supplying a limited interaction interface to the compressor instead of the operating system’s full set of system calls. VxUnZIP achieves these goals with a virtualized execution environment, VX32, that combines two different isolation mechanisms. To prevent untrusted code from reading or writing memory locations outside the sandbox, VX32 uses the IA-32 legacy segmented memory addressing mode to restrict the locations available to it. This requires operating system support to modify the local descriptor table (LDT), and segmentation is not supported in the 64-bit mode of newer processors, though VX32 can still work in 32-bit compatibility mode. To control which instructions the untrusted code executes (to protect for instance against unsafe indirect jumps or instructions that modify the segment registers), VX32 uses dynamic translation, rewriting code one fragment of a time into a cache and interpreting indirect jumps.

The author of VXA was not aware of PittSFIeld at the time it was designed, but to examine whether PittSFIeld would be a suitable replacement for VX32, we used it to protect the execution of the six VXA decompression modules demonstrated in [11]. We used VX32’s virtual C library rather than the one used in Section 7; this required implementing VXA’s four virtual system calls (`read`, `write`, `_exit`, and `sbrk`). We also used VX32’s library of math functions, but compiled to use the x87-style floating point instructions supported by PittSFIeld rather than the SSE2 ones VX32 uses. The runtime overheads of VX32 and PittSFIeld are compared in Figure 6. Zlib and BZip2 are decompressors for the same general-purpose compression formats as the SPECint2000 gzip and bzip2 programs (which also include compression); JPEG and JPEG2000 are lossy image compression formats, and FLAC and Vorbis are lossless and lossy audio formats respectively. In each case the programs decompressed large single files of each format. To minimize I/O overhead, we made sure the input files were cached in memory, and sent the decompressed output to `/dev/null`; measurements are based on elapsed time. The system was the same Pentium 4 machine described in Section 7, except that VX32 uses a specially compiled version of GCC 4.0, and the native

	Zlib	BZip2	JPEG	JPEG2000	FLAC	Vorbis	Geom. Mean
VX32	1.006	0.975	1.034	1.283	0.954	0.948	1.028
PittSFIeld jump-only	1.238	1.018	1.134	1.114	1.142	1.239	1.145
PittSFIeld full	1.398	1.072	1.328	1.211	1.241	1.458	1.278

Figure 6: Run time ratios for VX32 and PittSFIeld on the VXA decompressors, compared to natively compiled decompressors.

build uses Debian's GCC 4.0 to match.

The occasional speedup of execution under VX32, also seen in [11], appears to result from increased instruction locality introduced by translating dynamic execution traces sequentially. (For instance, VX32 is faster than native execution in the FLAC example even though it executes more instructions, 97.5 billion compared to 96.0 billion.) These examples have also been tuned to minimize the number of indirect jumps: two frequently called functions were inlined. The measured overhead of PittSFIeld for the vxUnZIP examples is noticeably higher than that of VX32, but still not excessive (28% on average). PittSFIeld's overhead is also smaller when protecting only jumps (averaging 15%); this simulates the performance of combining PittSFIeld with VX32-like segment-based memory protection.

Some qualitative features also affect the choice between PittSFIeld and VX32. An advantage of VX32 is that it prevents the decompressor from reading memory outside its sandbox; though not as critical for security as preventing writes, this is useful to ensure decompressors are deterministic. Controlling reads is possible with SFI, but would significantly increase the technique's overhead. On the other hand, VX32's use of segment registers decreases its portability, including to future processors; conversely, VX32's use of SSE2 floating point currently keeps it from working on older processors, though the latter limitation is not fundamental. Arguably, PittSFIeld's simple, static approach and separate verification make it more trustworthy, but VX32 is not yet as mature as PittSFIeld, and it is significantly simpler than previous dynamic translation systems.

9 Formal Analysis

Having restricted ourselves to a separate, minimal verification tool as the guarantor of our technique's safety, we can devote more effort to analyzing and assuring ourselves of that component's soundness. Specifically, we have constructed a completely formal and machine-checked proof of the fact that our technique ensures the security policy it claims to. Though the security of a complete system of course depends on many factors, such a proof provides a concise and trustworthy summary of the key underlying principles. Formal theorem proving has a reputation for being arduous; we think the relative ease with which this proof was completed is primarily a testament to the simplicity of the technique to which it pertains.

We have constructed the proof using ACL2 [13]. ACL2 is a theorem-proving system that combines a restricted subset of Common Lisp, used to model a system, with a sophisticated engine for semi-automatically proving theorems about those models. We use the programming language (which is first-order and purely functional) to construct a simplified model of our verifier, and a simulator for the x86 instruction set. Then, we give a series of lemmas about the behavior of the model, culminating in the statement of the desired safety theorem. The lemmas are chosen to be sufficiently elementary that ACL2 can automatically prove each from the model and the preceding lemmas. The proof required less than two months of effort by a user with no previous experience with proof assistants (the first author). An experienced ACL2 user could likely have produced a more elegant proof in less time; our inexperience in choosing abstractions also made the going more difficult as the proof size increased. An example of a function from the executable model and a lemma we have proved about it are shown as the first two parts of Figure 7. A disadvantage of ACL2 compared to some other theorem provers is that its proofs cannot be automatically checked by a simpler proof checker. However, ACL2 has been well tested by other academic and industrial users, and its underlying logic is simple, so we still consider it trustworthy.

The precise statement of our final safety result appears as the bottom part of Figure 7. It is a correctness result about the verifier, modeled as a predicate `mem-sandbox-ok` on the state of the code region before execution: if the verifier approves the rewritten code, then for any inputs (modelled as the initial contents of registers and the data region), execution of the code will continue forever without performing an unsafe operation. (Unlike the real system, the model has no `exit()` function.) Note that the rewriter does not appear in the formal proof, so the proof makes no claims about it: for instance, we have not proved that the code produced by the rewriter has the same behavior as the original code. Though a statement like that could be formalized, it would require a number of additional hypotheses; in particular, because the rewriter changes the

```
(defun seq-reachable-rec (mem eip k)
 (if (zp k) (if (= eip (code-start)) 0 nil)
  (let ((kth-insn
        (kth-insn-from mem (code-start) k)))
   (or (and kth-insn (= eip kth-insn) k)
       (seq-reachable-rec mem eip (- k 1))))))
(defthm if-reach-in-k-then-bound-by-kth-insn
 (implies
    (and (mem-p mem) (natp k) (natp eip)
         (kth-insn-from mem (code-start) k)
         (seq-reachable-rec mem eip k))
    (<= eip (kth-insn-from mem
                           (code-start) k))))
(defthm safety
  (implies
    (and (mem-p mem) (mem-sandbox-ok mem)
         (addr-p eax) (addr-p ebx) (addr-p ecx)
         (addr-p edx) (addr-p esi) (addr-p edi)
         (addr-p ebp) (addr-p esp)
         (addr-p eflags) (data-region-p ebp))
    (consp
     (step-for-k
       (x86-state (code-start) eflags eax ebx
                  ecx edx esi edi ebp esp mem)
        k))))
```

Figure 7: From top to bottom, a typical function definition, a typical lemma, and the final safety result from our formal ACL2 proof. `seq-reachable-rec` is a recursive procedure that checks whether the instruction at location `eip` is among the first k instructions reachable from the beginning of the sandboxed code region in a memory image `mem`. The lemma states that if `eip` is among the first k instructions, then its address is at most that of the kth instruction. The safety theorem states that if a memory image `mem` passes the verifier `mem-sandbox-ok`, then whatever the initial state of the registers, execution can proceed for any number of steps (the free variable k) without causing a safety violation (represented by a `nil` return value from `step-for-k`, which would not satisfy the predicate `consp`).

address of instructions, code that say examined the numeric values of function pointers would not behave identically.

One aspect of the proof to note is that it deals with a subset of the instructions handled by the real tool: this applies both to which instructions are allowed by the simulated verifier, and to which can be executed by the x86 simulator. The subset used in the current version of the proof appears in Figure 8. The instructions were chosen to exercise all of the aspects of the security policy; for instance, `jmp *%ebx` is included to demonstrate an indirect jump. Though small compared to the number of instructions allowed by the real tool, this set is similar to the instruction sets used in recent similar proofs [2, 29]. We constructed the proof by beginning with a minimal set of instructions and then adding additional ones: adding a new instruction similar to an existing one required few changes, but additions that required

```
nop          mov addr, %eax     xchg %eax, %ebx
inc %eax     mov %eax, addr     xchg %eax, %ebp
jmp addr     and $immed, %ebx   mov %eax, (%ebx)
jmp *%ebx    and $immed, %ebp   mov %eax, (%ebp)
```

Figure 8: List of instructions in the subset considered in the proof of Section 9.

a more complex safety invariant often involved extensive modifications. The simulator is structured so that an attempt to execute any un-modelled instruction causes an immediate failure, so safety for a program written in the subset that is treated in the proof extends to the complete system. A related concern is whether the simulated x86 semantics match those of a real processor: though the description of the subset used in the current proof can be easily checked by hand, this would be impractical for a more complete model. To facilitate proofs like ours in the future, as well as for applications such a foundational proof-carrying code (see Section 10.6), it should be possible to generate a description of the encoding and semantics of instructions from a concise, declarative, and proof-environment-neutral specification.

In total, the proof contains approximately 60 function definitions and 170 lemmas, over about 2400 lines of ACL2 code. The description of the model and the statement of the safety result require about 500 lines; assuming ACL2's verification is correct, only this subset must be trusted to be convinced of the truth of the result. The technical details of the proof are straightforward and rather boring; for space reasons, we do not discuss them further here. Interested readers are referred to a companion technical report [17]; the proof in its machine-checkable form is also available from the PittSFIeld project website.

10 Related work

This section compares our work with previous implementations of SFI, and with other techniques that ensure memory safety or isolation including code rewriting, dynamic translation, and low-level type systems. It also distinguishes the *isolation* provided by SFI from the *subversion protection* that some superficially similar techniques provide.

10.1 Other SFI implementations

Binary sandboxing was introduced as a technique for fault-isolation by Wahbe, Lucco, Anderson, and Graham [27]. The basic features of their approach were described in Sections 2 and 4. Wahbe et al. mention in a footnote that their technique would not be applicable to architectures like the x86 without some other technique to restrict control flow, but then drop the topic.

Subsequent researchers generally implemented a restriction on control flow for CISC architectures by collecting an explicit list of legal jump targets. The best example of such a system is Small and Seltzer's MiSFIT [25], an assembly-language rewriter designed to isolate faults in C++ code for an extensible operating system. MiSFIT generates a hash table from the set of legal jump targets in a program, and redirects indirect jumps through code that checks that the target appears in the table. Function return addresses are also stored on a separate, protected stack. Because control flow is prevented from jumping into the middle of them, the instruction sequences to sandbox memory addresses don't require a dedicated register, though MiSFIT does need to spill to the stack to obtain a scratch register in some cases. A less satisfying aspect of MiSFIT is its trust model. The rewriting engine and the code consumer must share a secret, which the rewriter uses to sign the generated code, and MiSFIT relies on the compiler to correctly manage the stack and to produce only safe references to call frames. Besides the trustworthiness problems of C compilers related to their complexity and weak specification (as exemplified by the attack against MiSFIT shown in Figure 1), this approach also requires something like a public-key certificate infrastructure for code producers, introducing problems of reputation to an otherwise objective question of code behavior.

Erlingsson and Schneider's SASI tool for the x86 [10] inserts code sequences very similar to MiSFIT's, except that its additions are pure checks that abort execution if an illegal operation is attempted, and otherwise fall through to the original code, like PittSFIeld's 'check' mode. In particular, the SASI tool is similar to MiSFIT in its use of a table of legal jump targets, and its decision to trust the compiler's manipulation of the stack. Lu's C+J system [16] also generates a table of legal jump destinations, but the indices into the table are assigned sequentially at translation time, so there is no danger of collision.

The Omniware virtual machine [3], on which Wahbe and Lucco worked after the classic paper, uses SFI in translating from a generic RISC-like virtual machine to a variety of architectures, including the x86. The Omniware VM implemented extensive compiler-like optimizations to reduce the overhead of sandboxing checks, achieving average overheads of about 10% on selected SPEC92 benchmarks. However, the focus of the work appears to have been more on performance and portability than on security; available information on the details of the safety checks, especially for the x86, is sparse. In a patent [28] Wahbe and Lucco disclose that later versions of the system enforced more complex, page-table like memory permissions, but give no more details of the x86 implementation.

As far as we know, our work described in Section 9 was the first machine-checked or completely formalized soundness proof for an SFI technique or implementation. Necula and Lee [20] proved the soundness of SFI as applied to particular programs, but not in general, and only in the context of simple packet filters. In work concurrent with ours, Abadi et al. ([2], see Section 10.3 for discussion) give a human-readable prose proof for the safety of a model of their CFI system, which is similar to SFI. In work subsequent to our proof (first described in [18]), Winwood and Chakravarty developed a machine-checked safety proof in Isabelle/HOL for a model of an SFI-like rewriting technique applicable to RISC architectures [29]. To avoid having to move instructions, their approach overwrites indirect jump instructions with direct jumps of the same size to a trusted dispatcher. Unfortunately, this puts a 2MB limit on the size of binaries to which their technique is applicable: for instance, they were able to rewrite only a subset of the SPECint2000 suite.

10.2 Isolation and preventing subversion

In general, a security failure of a system occurs when an attacker chooses input that causes code to perform differently than its author intended, and the subverted code then uses privileges it has to perform an undesirable action. Such an attack can be prevented either by preventing the code's execution for being subverted, or by isolating the vulnerable code so that even if subverted, it can still cannot take an undesirable action. Many security techniques are based on the prevention of subversion: for instance, ensuring that procedure calls always return to their call sites, even if the stack has been modified by a buffer overrun. SFI, by contrast, is fundamentally a technique for isolating one part of a program from another. To function as a security technique, this isolation must be used to support a design that divides a system into more and less trusted components, and restricts the interactions between the two. Examples of such designs include the device driver and network server isolation techniques discussed in Section 1, and the untrusted VXA decompressors of Section 8.

Incidentally, SFI subsumes some mechanisms that have been suggested as measures to make program subversion more difficult. For instance, PittSFIeld prohibits execution of code on the stack and reduces the number of possible targets of an overwritten function pointer. However, these side-effects should not be confused with the intended isolation policy. SFI does not provide general protection against attacks on the untrusted code; it simply contains those attacks within the component.

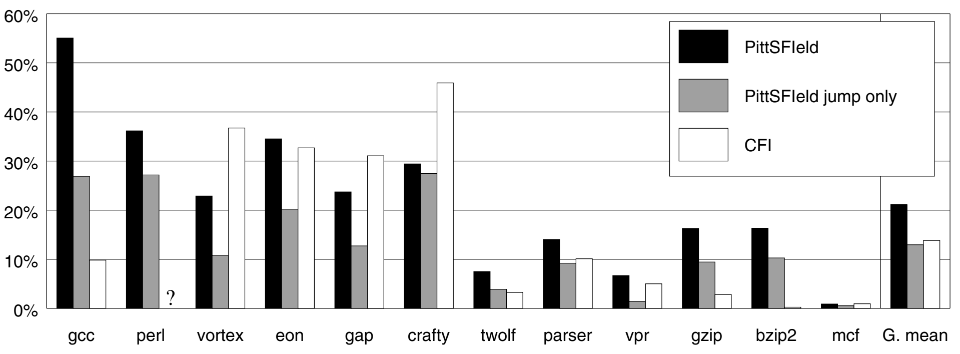


Figure 9: Runtime overheads for PittSFIeld in the default mode (black bars), PittSFIeld in jump-only mode (gray bars), and CFI (white bars) for the SPECint2000 benchmarks. PittSFIeld results are the same as those in Figure 4, but not broken down by cause. CFI results are taken from Figure 4 of [1], which does not include results for Perl. Because these were separate experiments with other variables not held constant, care should be used in comparing them directly.

10.3 CFI

In concurrent work [1], the Gleipnir project at Microsoft Research has investigated a binary-rewriting security technique called Control-Flow Integrity, or CFI. As suggested by the name, CFI differs from SFI in focusing solely on constraining a program's jumps: in the CFI implementation, each potential jump target is labelled by a 32-bit value encoded in a no-op instruction, and an indirect jump checks for the presence of an appropriate tag before transferring control. This approach gives finer control of jump destinations than the SFI techniques of Wahbe et al., or PittSFIeld, though the ideal precision could only be obtained with a careful static analysis of, for instance, which function pointers might be used at which indirect call sites. In the basic presentation, CFI relies on an external mechanism (such as hardware) to prevent changes to code or jumps to a data region, but it can also be combined with inserted memory-operation checks, as in SFI, to enforce these constraints simultaneously.

In the control-flow-only use, CFI has overheads ranging from 0 to 45% on a Pentium 4; the wide variation presumably results from a large overhead on indirect jumps combined with little overhead on any other operation. By comparison, PittSFIeld imposes a smaller overhead on jumps, but significant additional overheads on other operations. Figure 9 compares the overheads reported in [1] with those for PittSFIeld from Figure 4. Because a different C compiler, library, and hardware were used, caution should be used in directly comparing the PittSFIeld and CFI results, but overall the average overheads of the tools can be seen to be comparable. The benchmark labelled "?", 253.perlbmk, was omitted from [1] because of last-minute implementation difficulties [9], and is excluded from the CFI average.

Like PittSFIeld, CFI performs a separate verification to enforce proper rewriting at load time, so the compiler and binary rewriting infrastructure need not be trusted. The CFI authors have written a human-checked proof [2] that a CFI-protected program will never make unsafe jumps, even in the presence of arbitrary writes to data memory. However, the proof is formulated in terms of a miniature RISC architecture whose encoding is not specified. This is somewhat unsatisfying, as the safety of the real CFI technique is affected in subtle ways by the x86 instruction encoding (for instance, the possibility that the immediate value used in the comparison at a jump site might be itself interpreted as a safe jump target tag.)

10.4 Static C safety mechanisms

Another class of program rewriting tools (often implemented as compiler modifications) are focused on ensuring fairly narrow security policies, for instance that the procedure return address on the stack is not modified [6]. Such tools can be very effective in their intended role, and tend to have low overheads, but they do not provide protection against more esoteric subversion attacks. They also do not provide isolation between components, and are not intended for untrusted code. They could, however, be used in conjunction with SFI if both isolation and protection from subversion are desired.

10.5 Dynamic translation mechanisms

Several recent projects has borrowed techniques from dynamic optimization to rewrite programs on the fly; such techniques allow for fine control of program execution, as well as avoiding the difficulties of static binary rewriting. Valgrind [22] is a powerful framework for dynamic rewriting of Linux/x86 programs, which is best known for Purify-like memory checking, but can also be adapted to a number of other purposes. Valgrind's rewriting uses a simplified intermediate language, sacrificing performance for ease of development of novel applications. A research tool with a more security-oriented focus is Scott and Davidson's Strata [24]; it has achieved lower overheads (averaging about 30%) while enforcing targeted security policies such as system call interception. A similar but even higher performance system is Kiriansky et al.'s program shepherding [15], based on the DynamoRIO dynamic translation system. Their work concentrates on preventing attacks on a program's control flow, as an efficient and transparent means to prevent stack- and function-pointer-smashing vulnerabilities from being exploited. The VX32 system described in Section 8 also falls into this category. A disadvantage of dynamic techniques is that they are inherently somewhat complex and difficult to reason about, relative to a comparable static translation.

10.6 Low-level type safety

Recent research on verifiable low-level program representations has concentrated most strongly on static invariants, such as type systems. For instance, typed assembly language [19] can provide quickly checkable, fine-grained safety properties for a sublanguage of x86 assembly, but requires that the original program be written in a type-safe language. Type inference can also be used to transform C code into a type-safe program with a minimal set of dynamic checks, as in the CCured system [5]. Because they can constrain writes to a occur on specific objects, type-based safety properties are generally quite effective at preventing subversion attacks that overwrite function pointers.

Proof-carrying code [21] represents a more general framework for software to certify its own trustworthiness. Most work on PCC has focused on type-like safety properties, but under the banner of foundational PCC [4], efforts have been made to place proofs on a more general footing, using fully general proof languages that prove safety with respect to concrete machine semantics. This approach seems to carry the promise, not yet realized, of allowing any safe rewriting to certify its safety properties to a code consumer. For instance, one could imagine using the lemmas from the proof of Section 9 as part of a foundational safety proof for a PittSFIeld-rewritten binary. It is unclear, however, if any existing foundational PCC systems are flexible enough to allow such a proof to be used.

11 Conclusion

We have argued that software-based fault isolation can be a practical tool in constructing secure systems. Using a novel technique of artificially enforcing alignment for jump targets, we show how a simple sandboxing implementation can be constructed for an architecture with variable-length instructions like the x86. We give two new optimizations, which along with previously known ones minimize the runtime overhead of the technique, and argue for the importance of an architecture that includes separate verification. We have constructed a machine-checked soundness proof of our technique, to further enhance our confidence in its security. Finally, we have constructed an implementation of our technique which demonstrates separate verification and is scalable to large and complex applications. The performance overhead of the technique, as measured on both standard compute-intensive benchmarks and a realistic data compression application, is relatively low. Though some related techniques have lower runtime overheads, and others can offer additional security guarantees, SFI's combination of simplicity and performance is a good match for many uses.

Acknowledgements

Bryan Ford provided us with the VXA infrastructure used in the case study of Section 8, and Mihai Budiu and Úlfar Erlingsson provided results for Figure 9 and answered other questions about CFI. Members of the MIT PDOS and PAG groups, and the Harvard programming languages and compilers groups, provided a number of helpful suggestions. The first author is supported by a National Defense Science and Engineering Graduate Fellowship.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*, pages 340–353, Alexandria, VA, November 7–11, 2005.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A theory of secure control flow. In *Proceedings of the 7th International Conference on Formal Engineering Methods (ICFEM'05)*, pages 111–124, Manchester, UK, November 1–4, 2005.

[3] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and language-independent mobile programs. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 127–136, Philadelphia, PA, May 21–24, 1996.

[4] Andrew W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS'01)*, page 247, June 16–19, 2001.

[5] Jeremy Condit, Mathew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in

the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 232–244, San Diego, CA, USA, June 9–11, 2003.

[6] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, Austin, Texas, January 28–29, 1998. USENIX Association.

[7] Peter Deutsch and Charles A. Grant. A flexible measurement tool for software systems. In *Information Processing 71: Proceedings of IFIP Congress 71*, pages 320–326, Ljubljana, Yugoslavia, August 23–28, 1971.

[8] Daniel C. DuVarney, Sandeep Bhatkar, and V.N. Venkatakrishnan. SELF: a transparent security extension for ELF binaries. In *Proceedings of the 2003 New Security Paradigms Workshop*, pages 29–38, Ascona, Switzerland, August 18–21, 2003.

[9] Úlfar Erlingsson. Personal communication, May 2006.

[10] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 New Security Paradigms Workshop*, pages 87–95, Caledon Hills, ON, Canada, September 22–24 1999.

[11] Bryan Ford. VXA: A virtual architecture for durable compressed archives. In *4th USENIX Conference on File and Storage Technologies*, pages 295–308, San Francisco, CA, December 14–16, 2005.

[12] Andreas Gal, Christian W. Probst, and Michael Franz. A denial of service attack on the Java bytecode verifier. Technical Report 03-23, University of California, Irvine, School of Information and Computer Science, November 2003.

[13] Matt Kaufmann and J Strother Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.

[14] Douglas Kilpatrick. Privman: A library for partitioning applications. In *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)*, pages 273–284, San Antonio, TX, USA, June 12–14, 2003.

[15] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, San Francisco, California, August 7–9, 2002. USENIX Association.

[16] Fei Lu. C Plus J software architecture. Undergraduate thesis, Shanghai Jiaotong University, June 2000. English summary at `http://www.cs.jhu.edu/~flu/cpj/CPJ_guide.htm`.

[17] Stephen McCamant. A machine-checked safety proof for a CISC-compatible SFI technique. Technical Report 2006-035, MIT Computer Science and Artificial Intelligence Laboratory, May 2006.

[18] Stephen McCamant and Greg Morrisett. Efficient, verifiable binary sandboxing for a CISC architecture. Technical Report 2005-030, MIT Compter Science and Artificial Intelligence Lab, May 2005. (also MIT LCS TR #988).

[19] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Second ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, USA, May 1, 1999.

[20] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 229–243, Seattle, Washington, October 28–31, 1996.

[21] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 333–344, Montreal, Canada, June 17–19 1998.

[22] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, Boulder, Colorado, USA, July 13, 2003.

[23] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, pages 231–242, Washington, D.C., August 6–8, 2003. USENIX Association.

[24] Kevin Scott and Jack Davidson. Safe virtual execution using software dynamic translation. In *Proceedings of the 2002 Annual Computer Security Application Conference*, pages 209–218, Las Vegas, Nevada, December 9–13, 2002.

[25] Christopher Small. MiSFIT: A tool for constructing safe extensible C++ systems. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, pages 174–184, Portland, OR, USA, June 16–20 1997.

[26] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 207–222, Bolton Landing, NY, October 20–22, 2003.

[27] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, USA, December 5–8, 1993.

[28] Robert S. Wahbe and Steven E. Lucco. Methods for safe and efficient implementations of virtual machines. U.S. Patent 5,761,477, June 1998. Assigned to Microsoft Corporation.

[29] Simon Winwood and Manuel M. T. Chakravarty. Secure untrusted binaries - provably! In *Workshop on Formal Aspects in Security and Trust (FAST 2005)*, pages 171–186, Newcastle upon Tyne, U.K., July 18–19, 2005.

SigFree: A Signature-free Buffer Overflow Attack Blocker

[1]Xinran Wang [2]Chi-Chun Pan [2]Peng Liu [1,2]Sencun Zhu
[1]*Department of Computer Science and Engineering*
[2]*College of Information Sciences and Technology*
The Pennsylvania State University, University Park, PA 16802
{xinrwang, szhu}@cse.psu.edu; {cpan, pliu}@ist.psu.edu

Abstract

We propose SigFree, a realtime, signature-free, out-of-the-box, application layer blocker for preventing buffer overflow attacks, one of the most serious cyber security threats. SigFree can filter out code-injection buffer overflow attack messages targeting at various Internet services such as web service. Motivated by the observation that buffer overflow attacks typically contain executables whereas legitimate client requests never contain executables in most Internet services, SigFree blocks attacks by detecting the presence of code. SigFree first blindly dissembles and extracts instruction sequences from a request. It then applies a novel technique called *code abstraction*, which uses data flow anomaly to prune useless instructions in an instruction sequence. Finally it compares the number of useful instructions to a threshold to determine if this instruction sequence contains code. SigFree is signature free, thus it can block new and unknown buffer overflow attacks; SigFree is also immunized from most attack-side code obfuscation methods. Since SigFree is transparent to the servers being protected, it is good for economical Internet wide deployment with very low deployment and maintenance cost. We implemented and tested SigFree; our experimental study showed that SigFree could block all types of code-injection attack packets (above 250) tested in our experiments. Moreover, SigFree causes negligible throughput degradation to normal client requests.

1 Introduction

Throughout the history of cyber security, buffer overflow is one of the most serious vulnerabilities in computer systems. Buffer overflow vulnerability is a root cause for most of the cyber attacks such as server breaking-in, worms, zombies, and botnets. Buffer overflow attacks are the most popular choice in these attacks, as they provide substantial control over a victim host [37].

"A buffer overflow occurs during program execution when a fixed-size buffer has had too much data copied into it. This causes the data to overwrite into adjacent memory locations, and, depending on what is stored there, the behavior of the program itself might be affected." [34] (Note that the buffer could be in stack or heap.) Although taking a broader viewpoint, buffer overflow attacks do not always carry code in their attacking requests (or packets)[1], code-injection buffer overflow attacks such as stack smashing count for probably most of the buffer overflow attacks that have happened in the real world.

Although tons of research has been done to tackle buffer overflow attacks, existing defenses are still quite limited in meeting four highly-desired requirements: (R1) *simplicity* in maintenance; (R2) *transparency* to existing (legacy) server OS, application software, and hardware; (R3) *resiliency* to obfuscation; (R4) economical Internet wide deployment. As a result, although several very secure solutions have been proposed, they are not pervasively deployed, and a considerable number of buffer overflow attacks continue to succeed on a daily basis.

To see how existing defenses are limited in meeting these four requirements, let us break down the existing buffer overflow defenses into six classes which we will review shortly in Section 2: (1A) Finding bugs in source code. (1B) Compiler extensions. (1C) OS modifications. (1D) Hardware modifications. (1E) Defense-side obfuscation [11,28]. (1F) Capturing code running symptoms of buffer overflow attacks [21,37,43,55]. (Note that the above list does not include binary code analysis based defenses which we will address shortly.) We may briefly summarize the limitations of these defenses in terms of the four requirements as follows. (a) Class 1B, 1C, 1D, and 1E defenses may cause substantial changes to existing (legacy) server OSes, application softwares, and hardwares, thus they are not transparent. Moreover, Class 1E defenses generally cause processes to be terminated. As a result, many businesses do not view these changes and the process termination overhead as economical deployment. (b) Class 1F defenses can be very secure, but they either suffer from significant runtime overhead or need special auditing or diagnosis facilities which are not commonly available in commercial services. As a result, Class 1F defenses have limited transparency and potential for economical deployment. (c) Class 1A defenses need source code, but source code is unavailable to many legacy applications.

Besides buffer overflow defenses, worm signatures can be generated and used to block buffer overflow attack packets [29, 42, 47]. Nevertheless, they are also limited in meeting the four requirements, since they either rely on signatures, which introduce maintenance overhead, or are not very resilient to attack-side obfuscation.

To overcome the above limitations, in this paper we propose SigFree, a realtime buffer overflow attack blocker, to protect Internet services. The idea of SigFree is motivated by an important observation that "the nature of communication to and from network services is predominantly or exclusively data and not executable code." [15]. In particular, as summarized in [15], (a) on Windows platforms, most web servers (port 80) accept data only; remote access services (ports 111, 137, 138, 139) accept data only; Microsoft SQL Servers (port 1434) accept data only; workstation services (ports 139 and 445) accept data only. (b) On Linux platforms, most Apache web servers (port 80) accept data only; BIND (port 53) accepts data only; SNMP (port 161) accepts data only; most Mail Transport (port 25) accepts data only; Database servers (Oracle, MySQL, PostgreSQL) at ports 1521, 3306 and 5432 accept data only.

Since remote exploits are typically executable code, this observation indicates that if we can precisely distinguish (service requesting) messages that contain code from those that do not contain any code, we can protect most Internet services (which accept data only) from code-injection buffer overflow attacks by blocking the messages that contain code.

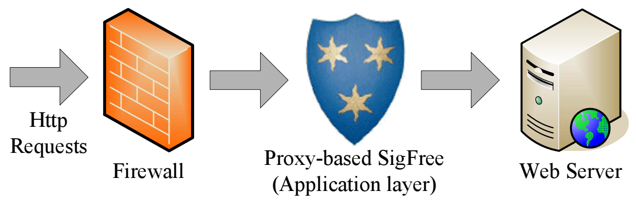


Figure 1: SigFree is an application layer blocker between the web server and the corresponding firewall.

Accordingly, SigFree (Figure 1) works as follows. SigFree is an application layer blocker that typically stays between a service and the corresponding firewall. When a service requesting message arrives at SigFree, SigFree first uses a new $O(N)$ algorithm, where N is the byte length of the message, to disassemble and distill all possible instruction sequences from the message's payload, where every byte in the payload is considered as a possible starting point of the code embedded (if any). However, in this phase some data bytes may be mistakenly decoded as instructions. In phase 2, SigFree uses a novel technique called *code abstraction*. Code abstraction first uses data flow anomaly to prune useless instructions in an instruction sequence, then compares the number of useful instructions to a threshold to determine if this instruction sequence (distilled in phase 1) contains code.

We have implemented a SigFree prototype as a proxy to protect web servers. Our empirical study shows that there exists clean-cut "boundaries" between code-embedded payloads and data payloads when our code-data separation criteria are applied. We have identified the "boundaries" (or thresholds) and been able to detect/block all 50 attack packets generated by Metasploit framework [4], all 200 polymorphic shellcode packets generated by two well-known polymorphic shellcode engine ADMmutate [40] and CLET [23], and worm Slammer, CodeRed and a CodeRed variation, when they are well mixed with various types of data packets. Also, our experiment results show that the throughput degradation caused by SigFree is negligible.

The merits of SigFree are summarized below. They show that SigFree has taken a main step forward in meeting the four requirements aforementioned.

⊙ *SigFree is signature free, thus it can block new and unknown buffer overflow attacks*

⊙ Without relying on string-matching, *SigFree is immunized from most attack-side obfuscation methods.*

⊙ *SigFree uses generic code-data separation criteria instead of limited rules.* This feature separates SigFree from [15], an independent work that tries to detect code-embedded packets.

⊙ *Transparency.* SigFree is a out-of-the-box solution that requires no server side changes.

⊙ *SigFree has negligible throughput degradation.*

⊙ *SigFree is an economical deployment with very low maintenance cost*, which can be well justified by the aforementioned features.

The rest of the paper is organized as follows. In Section 2, we summarize the work related to ours. In Section 3, we give an overview of SigFree. In Sections 4 and 5, we introduce the instruction sequence distiller component and the instruction sequence analyzer component of SigFree, respectively. In Section 6, we show our experimental results. Finally, we discuss some research issues in Section 7 and conclude the paper in Section 8.

2 Related Work

SigFree is mainly related to three bodies of work. [Category 1:] prevention/detection techniques of buffer overflows; [Category 2:] worm detection and signature generation. [Category 3:] machine code analysis for security purposes. In the following, we first briefly review Category 1 and Category 2 which are less close to SigFree. Then we will focus on comparing SigFree with Category 3.

2.1 Prevention/Detection of Buffer Overflows

Existing prevention/detection techniques of buffer overflows can be roughly broken down into six classes:

Class 1A: Finding bugs in source code. Buffer overflows are fundamentally due to programming bugs. Accordingly, various bug-finding tools [14, 24, 51] have been developed. The bug-finding techniques used in these tools, which in general belong to static analysis, include but not limited to model checking and bugs-as-deviant-behavior. Class 1A techniques are designed to handle source code only, and they do not ensure completeness in bug finding. In contrast, SigFree handles machine code embedded in a request (message).

Class 1B: Compiler extensions. "If the source code is available, a developer can add buffer overflow detection automatically to a program by using a modified compiler." [34] Three such compilers are StackGuard [22], ProPolice [2], and Return Address Defender (RAD) [18]. In addition, Smirnov and Chiueh proposed compiler DIRA [49] can detect control hijacking attacks, identify the malicious input and repair the compromised program. Class 1B techniques require the availability of source code. In contrast, SigFree does not need to know any source code.

Class 1C: OS modifications. Modifying some aspects of the operating system may prevent buffer overflows such as Pax [9], LibSafe [10] and e-NeXsh [48]. Class 1C techniques need to modify the OS. In contrast, SigFree does not need any modification of the OS.

Class 1D: Hardware modifications. A main idea of hardware modification is to store all return addresses on the processor [41]. In this way, no input can change any return address.

Class 1E: Defense-side obfuscation. Address Space Layout Randomization (ASLR) is a main component of PaX [9]. Bhatkar and Sekar [13] proposed a comprehensive address space randomization scheme. Address-space randomization, in its general form [13], can detect exploitation of all memory errors. Instruction set randomization [11, 28] can detect all code injection attacks. Nevertheless, when these approaches detect an attack, the victim process is typically terminated. "Repeated attacks will require repeated and expensive application restarts, effectively rendering the service unavailable." [37]

Class 1F: Capturing code running symptoms of buffer overflow attacks. Fundamentally, buffer overflows are a code running symptom. If such unique symptoms can be precisely captured, all buffer overflows can be detected. Class 1B, Class 1C and Class 1E techniques can capture some - but not all - of the running symptoms of buffer overflows. For example, accessing non-executable stack segments can be captured by OS modifications; compiler modifications can detect return address rewriting; and process crash is a symptom captured by defense-side obfuscation. To achieve 100% coverage in capturing buffer overflow symptoms, dynamic dataflow/taint analysis/program shepherding techniques were proposed in Vigilante [21], TaintCheck [43], and [30]. They can detect buffer overflows during runtime. However, it may cause significant runtime overhead (e.g., 1,000%). To reduce such overhead, another type of Class 1F techniques, namely post-crash symptom diagnosis, has been developed in Covers [37] and [55]. Post-crash symptom diagnosis extracts the 'signature' after a buffer overflow attack is detected. Recently, Liang and Sekar proposed ARBOR [36] which can automatically generate vulnerability-oriented signatures by identifying characteristic features of attacks and using program context. Moreover, ARBOR automatically invokes the recovery actions. Class 1F techniques can block both the attack requests that contain code and the attack requests that do not contain any code, but they need the signatures to be firstly generated. Moreover, they either suffer from significant runtime overhead or need special auditing or diagnosis facilities which are not commonly available in commercial services. In contrast, although SigFree could not block the attack requests that do not contain any code, SigFree is signature-free and does not need any changes to real world services. We will investigate the integration of SigFree with Class 1F techniques in our future work.

2.2 Worm Detection and Signature Generation

Because buffer overflows are a key target of worms when they propagate from one host to another, SigFree is related to worm detection. Based on the nature of worm infection symptoms, worm detection techniques can be broken down into three classes: [Class 2A] techniques use such macro symptoms as Internet background radiation (observed by network telescopes) to raise early warnings of Internet wide worm infection [44]. [Class 2B] techniques use such local traffic symptoms as content invariance, content prevalence and address dispersion to generate worm signatures and/or block worms. Some examples of Class 2B techniques are Earlybird [47], Autograph [29], Polygraph [42], and TRW [27]. [Class 2C] techniques use worm code running symptoms to detect worms. It is not surprising that Class 2C techniques are exactly Class 1F techniques. Some example Class 2C techniques are Shield [52], Vigilante [21], COVERS [37]. [Class 2D] techniques use anomaly detection on packet payload to detect worms and generate signature. Wang and Stolfo [54] first proposed Class 2D techniques called PAYL. PAYL is first trained with normal network flow traffic and then uses some byte-level statistical measures to detect exploit code. Recently, Wang et al. [53] proposed new features of PAYL that based on ingress/egress anomalous payload correlation to detect new worms and automatically generate signatures. FLIPS [39] uses PAYL [54] to detect anomalous inputs. If the anomaly is confirmed by a detector, a content-based signature is generated.

Class 2A techniques are not relevant to SigFree. Class 2C techniques are already discussed. Class 2D techniques could be evaded by statistically mimics normal traffic [31]. Class 2B techniques rely on signatures, while SigFree is signature-free. Class 2B techniques focus on identifying the unique bytes that a worm packet must carry, while SigFree focuses on determining if a packet contains code or not. Exploiting the content invariance property, Class 2B techniques are typically not very resilient to obfuscation. In contrast, SigFree is immunized from most attack-side obfuscation methods.

2.3 Machine Code Analysis for Security Purposes

Although source code analysis has been extensively studied (see Class 1A), in many real world scenarios source code is not available and the ability to analyze binaries is desired. Machine code analysis has three main security purposes: (P1) malware detection, (P2) to analyze obfuscated binaries, and (P3) to identify and analyze the code contained in buffer overflow attack packets. Along purpose P1, Chritodorescu and Jha [16] proposed static analysis techniques to detect malicious patterns in executables, and Chritodorescu et al. [17] exploited semantic heuristics to detect obfuscated malware. Along purpose P2, Lakhotia and Eric [35] used static analysis techniques to detect obfuscated calls in binaries, and Kruegel et al. [33] investigated disassembly of obfuscated binaries.

SigFree differs from P1 and P2 techniques in design goals. The purpose of SigFree is to see if a message contains code or not, instead of determining if a piece of code has malicious intent or not. (Note that SigFree does not check if the code contained in a message has malicious intent.) Due to this reason, SigFree is immunized from most attack-side obfuscation methods. Nevertheless, both the techniques in [33] and SigFree disassemble binary code, although their disassembly procedures are different. As will be seen, disassembly is not the kernel contribution of SigFree.

Fnord [2], the preprocessor of Snort IDS, identifies exploit code by detecting NOP sled. Toth and Kruegel [50] also aim at detecting NOP sled. They employed binary disassembly to find the sequence of execution instructions as an evidence of a NOP sled. However, Some attacks such as worm CodeRed do not include NOP sled and, as mentioned in [15], mere binary disassembly is not adequate. Moreover, polymorphic shellcode [23, 40] can bypass the detection for NOP instructions by using fake NOP zone. SigFree does not rely on the detection of NOP sled.

Finally, being generally a P3 technique, SigFree is most relevant to two P3 works [15, 32]. Kruegel et al. [32] innovatively exploited control flow structures to detect polymorphic worms. Unlike string-based signature matching, their techniques identify structural similarities between different worm mutations and use these similarities to detect more polymorphic worms. The implementation of their approach is resilient to a number of code transformation techniques. Although their techniques also handle binary code, they perform offline analysis. In contrast, SigFree is an online attack blocker. As such, their techniques and SigFree are complementary to each other with different purposes. Moreover, unlike SigFree, their techniques [32] may not be suitable to block the code contained in *every* attack packet, because some buffer overflow code is so simple that very little control flow information can be exploited.

Independent of our work, Chinchani and Berg [15] proposed a rule-based scheme to achieve the same goal of SigFree, that is, to detect exploit code in network flows. However, there is a fundamental difference between SigFree and their scheme [15]. Their scheme is rule-based, whereas SigFree is a *generic* approach which does not require any pre-known patterns. More specifically, their scheme [15] first tries to find certain pre-known instructions, instruction patterns or control flow structures in a packet. Then, it uses the found patterns and a data flow analysis technique called program slicing to analyze the packet's payload to see if the packet really contains code. Four rules (or cases) are discussed in their paper: Case 1 not only assumes the occurrence of the call/jmp instructions, but also expects the push instruction appears before the branch; Case 2 relies on the *interrupt* instruction; Case 3 relies on instruction *ret*; Case 4 exploits hidden branch instructions. Besides, they used a special rule to detect polymorphic exploit code which contains a loop. Although they mentioned that the above rules are initial sets and may require updating with time, it is always possible for attackers to bypass those pre-known rules. Moreover, more rules mean more overhead and longer latency in filtering packets. In contrast, SigFree exploits a different data flow analysis technique, which is much harder for exploit code to evade.

3 SigFree Overview

3.1 Basic Definitions and Notations

This section provides the definitions that will be used in the rest of the paper.

Definition 1 *(instruction sequence) An instruction sequence is a sequence of CPU instructions which has one and only one entry instruction and there exist at least one execution path from the entry instruction to any other instruction in this sequence.*

An instruction sequence is denoted as s_i, where i is the entry address of the instruction sequence. A fragment of a program in machine language is an instruction sequence, but an instruction sequence is not necessarily a fragment of a program. In fact, we may distill instruction sequences from any binary strings. This poses the fundamental challenge to our research goal. Figure 2 shows four instruction sequences distilled from a substring of a GIF file. These four instruction sequences are not fragments of a real program, although they may also be executed in a specific CPU. Below we call them *random instruction sequences*, whereas use the term *binary executable code* to refer to a fragment of a real program in machine language.

Definition 2 *(instruction flow graph) An instruction flow graph (IFG) is a directed graph $G = (V, E)$ where each node $v \in V$ corresponds to an instruction and each edge $e = (v_i, v_j) \in E$ corresponds to a possible transfer of control from instruction v_i to instruction v_j.*

Unlike traditional control flow graph (CFG), a node of an IFG corresponds to a single instruction rather than a basic block of instructions. To completely model the control flow of an instruction sequence, we further extend the above definition.

Definition 3 *(extended instruction flow graph) An extended instruction flow graph (EIFG) is a directed graph $G = (V, E)$ which satisfies the following properties: each node $v \in V$ corresponds to an instruction, an illegal instruction, or an external address; each edge $e = (v_i, v_j) \in E$ corresponds to a possible transfer of control from instruction v_i to instruction v_j, to illegal instruction v_j, or to an external address v_j.*

Accordingly, we name the types of nodes in an EIFG *instruction node*, *illegal instruction node*, and *external address node*.

The reason that we define IFG and EIFG is to model two special cases which CFG cannot model (the difference will be very evident in the following sections). First, in an instruction sequence, control may be transferred from an instruction node to an illegal instruction node. For example, in instruction sequence s_{08} in Figure 2, the transfer of control is from instruction "lods [ds:esi]" to an illegal instruction at address $0F$. Second, control may be transferred from an instruction node to an external address node. For example, instruction sequence s_{00} in Figure 2 has an instruction "jmp ADAAC3C2", which jumps to external address ADAAC3C2.

3.2 Attack Model

An attacker exploits a buffer overflow vulnerability of a web server by sending a crafted request, which contains a malicious payload. Figure 3 shows the format of a HTTP request. There are several HTTP request methods among which GET and POST are most often used by attackers. Although HTTP 1.1 does not allow GET to have a request body, some web servers such as Microsoft IIS still dutifully read the request-body according to the request-header's instructions (the CodeRed worm exploited this very problem).

The position of a malicious payload is determined by the exploited vulnerability. A malicious payload may be embedded in the Request-URI field as a query parameter. However, as the maximum length of Request-URI is limited, the size of a malicious payload, hence the behavior of such a buffer overflow attack, is constrained. It is more common that a buffer overflow attack payload is embedded in Request-Body of a POST method request. Technically, a malicious payload may also be embedded in Request-Header, although this kind of attacks have not been observed yet. In this work, we assume an attacker can use any request method and embed the malicious code in any field.

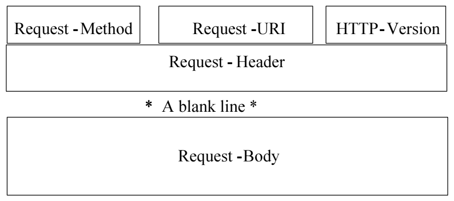


Figure 3: A HTTP Request. A malicious payload is normally embedded in Request-URI or Request-Body

3.3 Assumptions

In this paper, we focus on buffer overflow attacks whose payloads contain executable code in machine language, and we assume normal requests do not contain

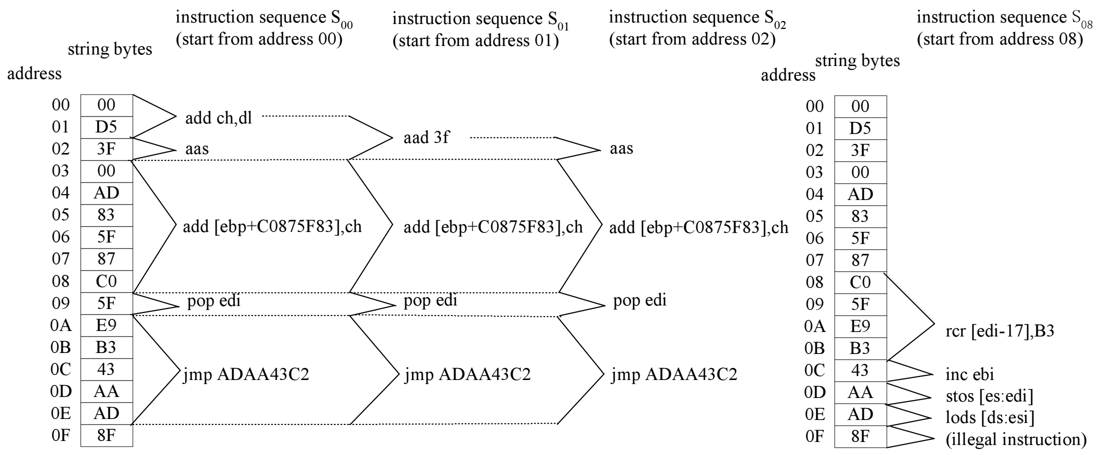

instruction sequence S_{00} (start from address 00)
instruction sequence S_{01} (start from address 01)
instruction sequence S_{02} (start from address 02)
instruction sequence S_{08} (start from address 08)
string bytes
address
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00 D5 3F 00 AD 83 5F 87 C0 5F E9 B3 43 AA AD 8F
add ch,dl
aas
add [ebp+C0875F83],ch
pop edi
jmp ADAA43C2
aad 3f
rcr [edi-17],B3
inc ebi
stos [es:edi]
lods [ds:esi]
(illegal instruction)

Figure 2: Instruction sequences distilled from a substring of a GIF file. We assign an address to every byte of the string. Instruction sequences s_{00}, s_{01}, s_{02} and s_{08} are distilled by disassembling the string from addresses 00, 01, 02 and 08, respectively.

executable machine code. A normal request may contain any data, parameters, or even a SQL statement. Note that although SQL statements are executable in the application level, they cannot be executed directly by a CPU. As such, SQL statements are not viewed as executable in our model. Application level attacks such as data manipulation and SQL injection are out of the scope.

Though SigFree is a generic technique which can be applied to any instruction set, for concreteness we assume the web server runs the Intel IA32 instruction set, the most popular instruction set running inside a web server today.

3.4 Architecture

Figure 4 depicts the architecture of SigFree and it is comprised of the following modules:

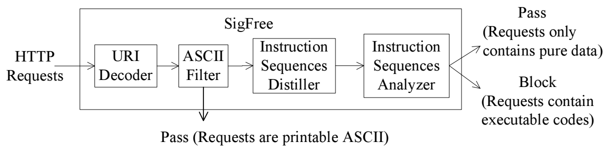

SigFree
HTTP Requests
URI Decoder
ASCII Filter
Instruction Sequences Distiller
Instruction Sequences Analyzer
Pass (Requests only contains pure data)
Block (Requests contain executable codes)
Pass (Requests are printable ASCII)

Figure 4: The architecture of SigFree

URI decoder. The specification for URLs [12] limits the allowed characters in a Request-URI to only a subset of the ASCII character set. This means that the query parameters of a request-URI beyond this subset should be encoded [12]. Because a malicious payload may be embedded in the request-URI as a request parameter, the first step of SigFree is to decode the request-URI.

ASCII Filter. Malicious executable code are normally binary strings. In order to guarantee the throughput and response time of the protected web system, if the query parameters of the request-URI and request-body of a request are both printable ASCII ranging from 20-7E in hex, SigFree allows the request to pass (In Section 7.2, we will discuss a special type of executable codes called alphanumeric shellcodes [45] that actually use printable ASCII) .

Instruction sequences distiller (ISD). This module distills all possible instruction sequences from the query parameters of Request-URI and Request-Body (if the request has one).

Instruction sequences analyzer (ISA). Using all the instruction sequences distilled from the instruction sequences distiller as the inputs, this module analyzes these instruction sequences to determine whether one of them is (a fragment of) a program.

4 Instruction Sequence Distiller

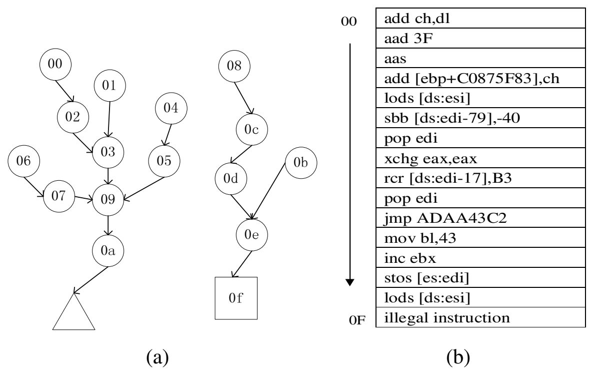
This section first describes an effective algorithm to distill instruction sequences from http requests, followed by several excluding techniques to reduce the processing overhead of instruction sequences analyzer.

4.1 Distilling Instruction Sequences

To distill an instruction sequence, we first assign an address to every byte of a request. Then, we disassemble the request from a certain address until the end of the request is reached or an illegal instruction opcode is encountered. There are two traditional disassembly algorithms: *linear sweep* and *recursive traversal* [38, 46]. The linear sweep algorithm begins disassembly at a certain address, and proceeds by decoding each encountered instruction. The recursive traversal algorithm also begins disassembly at a certain address, but it follows the control flow of instructions.

In this paper, we employ the recursive traversal algorithm, because it can obtain the control flow information during the disassembly process. Intuitively, to get all possible instruction sequences from a N-byte request, we simply execute the disassembly algorithm N times and each time we start from a different address in the request. This gives us a set of instruction sequences. The running time of this algorithm is $O(N^2)$.

One drawback of the above algorithm is that the same instructions are decoded many times. For example, instruction "pop edi" in Figure 2 is decoded many times by this algorithm. To reduce the running time, we design a memorization algorithm [20] by using a data structure, which is an EIFG defined earlier, to represent the instruction sequences. To distill all possible instruction sequences from a request is simply to create the EIFG for the request. An EIFG is used to represent all possible transfers of control among these instructions. In addition, we use an instruction array to represent all possible instructions in a request. To traverse an instruction sequence, we simply traverse the EIFG from the entry instruction of the instruction sequence and fetch the corresponding instructions from the instruction array. Figure 5 shows the data structure for the request shown in Figure 2. The details of the algorithm for creating the data structure are described in Algorithm 1. Clearly, the running time of this algorithm is $O(N)$, which is optimal as each address is traversed only once.



add ch,dl
aad 3F
aas
add [ebp+C0875F83],ch
lods [ds:esi]
sbb [ds:edi-79],-40
pop edi
xchg eax,eax
rcr [ds:edi-17],B3
pop edi
jmp ADAA43C2
mov bl,43
inc ebx
stos [es:edi]
lods [ds:esi]
illegal instruction

Figure 5: Data structure for the instruction sequences distilled from the request in Figure 2. (a) Extended instruction flow graph. Circles represent instruction nodes; triangles represent external addresses; rectangles represent illegal instructions. (b) The array of all possible instructions in the request.

4.2 Excluding Instruction Sequences

The previous step may output many instruction sequences at different entry points. Next we exclude some of them based on several heuristics. Here *excluding an instruction sequence means that the entry of this sequence is not considered as the real entry for the embedded code (if any).*

Algorithm 1 Distill all instruction sequences from a request

```
initialize EISG $G$ and instruction array $A$ to empty
for each address $i$ of the request do
    add instruction node $i$ to $G$
$i \leftarrow$ the start address of the request
while $i <=$ the end address of the request do
    $inst \leftarrow$ decode an instruction at $i$
    if $inst$ is illegal then
        $A[i] \leftarrow$ illegal instruction $inst$
        set type of node $i$ "illegal node" in $G$
    else
        $A[i] \leftarrow$ instruction $inst$
        if $inst$ is a control transfer instruction then
            for each possible target t of $inst$ do
                if target $t$ is an external address then
                    add external address node $t$ to $G$
                add edge e(node $i$, node $t$) to G
        else
            add edge e(node $i$, node $i + inst.length$) to G
        $i \leftarrow i + 1$
```

The fundamental rule in excluding instruction sequences is not to affect the decision whether a request contains code or not. This rule can be translated into the following technical requirements: if a request contains a fragment of a program, the fragment must be one of the remaining instruction sequences or a subsequence of a remaining instruction sequence, or it differs from a remaining sequence only by few instructions.

Step 1 If instruction sequence s_a is a subsequence of instruction sequence s_b, we exclude s_a. The rationale for excluding s_a is that if s_a satisfies some characteristics of programs, s_b also satisfies these characteristics with a high probability.

This step helps exclude lots of instruction sequences since many distilled instruction sequences are subsequences of the other distilled instruction sequences. For example, in Figure 5(a), instruction sequence s_{02}, which is a subsequence of instruction sequence s_{00}, can be excluded. Note that here we only exclude instruction sequence s_{02} rather than remove node v_{02}. Similarly, instruction sequences s_{03},s_{05}, s_{07}, s_{09},s_{0a},s_{0c},s_{0d} and s_{0e} can be excluded.

Step 2 If instruction sequence s_a merges to instruction sequence s_b after a few instructions (e.g., 4 in our experiments) and s_a is no longer than s_b, we exclude s_a. It is reasonable to expect that s_b will preserve s_a's characteristics.

Many distilled instruction sequences are observed to merge to other instructions sequences after a few instructions. This property is called self-repairing [38] in Intel IA-32 architecture. For example, in Figure 5(a) instruction sequence s_{01} merges to instruction sequence s_{00}

only after one instruction. Therefore, s_{01} is excluded. Similarly, instruction sequences s_{04}, s_{06} and s_{0b} can be excluded.

Step 3 For some instruction sequences, if we execute them, whatever execution path being taken, an illegal instruction is *inevitably reachable*. We say an instruction is inevitably reachable if two conditions holds. One is that there are no cycles (loops) in the EIFG of the instruction sequence; the other is that there are no external address nodes in the EIFG of the instruction sequence.

We exclude the instruction sequences in which illegal instructions are inevitably reachable, because causing the server to execute an illegal instruction is not the purpose of an buffer overflow attack (this assumption was also made by others [15, 32], implicitly or explicitly). Note that however the existence of illegal instruction nodes cannot always be used as a criteria to exclude an instruction sequence unless they are inevitably reachable; otherwise attackers may obfuscate their program by adding *non-reachable* illegal instructions.

Based on this heuristic, we can exclude instruction sequence s_{08} in Figure 5(a), since it will eventually execute an illegal instruction v_{0f}.

After these three steps, in Figure 5(a) only instruction sequence s_{00} is left for consideration in the next stage.

5 Instruction Sequences Analyzer

A distilled instruction sequence may be a sequence of random instructions or a fragment of a program in machine language. In this section, we propose two schemes to differentiate these two cases. Scheme 1 exploits the operating system characteristics of a program; Scheme 2 exploits the data flow characteristics of a program. Scheme 1 is slightly faster than Scheme 2, whereas Scheme 2 is much more robust to obfuscation.

5.1 Scheme 1

A program in machine language is dedicated to a specific operating system; hence, a program has certain characteristics implying the operating system on which it is running, for example calls to operating system or kernel library. A random instruction sequence does not carry this kind of characteristics. By identifying the call pattern in an instruction sequence, we can effectively differentiate a real program from a random instruction sequence.

More specifically, instructions such as "call" and "int 0x2eh" in Windows and "int 0x80h" in Linux may indicate system calls or function calls. However, since the op-codes of these call instructions are only one byte, even normal requests may contain plenty of these byte values. Therefore, using the number of these instructions as a criteria will cause a high false positive rate. To address this issue, we use a pattern composed of several instructions rather than a single instruction. It is observed that before these call instructions there are normally one or several instructions used to transfer parameters. For example, a "push" instruction is used to transfer parameters for a "call" instruction; some instructions that set values to registers al, ah, ax, or eax are used to transfer parameters for "int" instructions. These call patterns are very common in a fragment of a real program. Our experiments in Section 6 show that by selecting the appropriate parameters we can rather accurately tell whether an instruction sequence is an executable code or not.

Scheme 1 is fast since it does not need to fully disassemble a request. For most instructions, we only need to know their types. This saves lots of time in decoding operands of instructions.

Note that although Scheme 1 is good at detecting most of the known buffer overflow attacks, it is vulnerable to obfuscation. One possible obfuscation is that attackers may use other instructions to replace the "call" and "push" instructions. Figure 5.1 shows an example of obfuscation, where "call eax" instruction is substituted by "push J4" and "jmp eax". Although we cannot fully solve this problem, by recording this kind of instruction replacement patterns, we may still be able to detect this type of obfuscation to some extent.

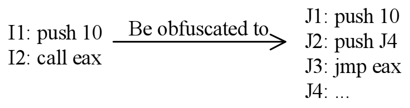


Figure 6: An obfuscation example. Instruction "call eax" is substituted by "push J4" and "jmp eax".

Another possible obfuscation is one which first encrypts the attack code and then decrypts it using a decryption routine during execution time [40]. This decryption routine does not include any calls, thus evading the detection of Scheme 1.

5.2 Scheme 2

Next we propose Scheme 2 to detect the aforementioned obfuscated buffer overflow attacks. Scheme 2 exploits the data flow characteristics of a program. Normally, a random instruction sequence is full of data flow anomalies, whereas a real program has few or no data flow anomalies. However, the number of data flow anomalies cannot be directly used to distinguish a program from a random instruction sequence because an attacker may obfuscate his program easily by introducing enough data flow anomalies.

In this paper, we use the detection of data flow anomaly in a different way called *code abstraction*. We

```
...
I1: mov eax,2
...
I2: mov eax,3
...
```

(a)

```
...
(ecx is undefined at
this point)
K1: mov eax,ecx
...
```

(b)

```
...
J1: mov eax,2
...
(ebx is undefined at
this point)
J2: mov eax,ebx
...
```

(c)

Figure 7: Data flow anomaly in execution paths. (a) define-define anomaly. Register eax is defined at I1 and then defined again at I2. (b) undefine-reference anomaly. Register ecx is undefined before K1 and referenced at K1 (c) define-undefine anomaly. Register eax is defined at J1 and then undefined at J2.

observe that when there are data flow anomalies in an execution path of an instruction sequence, some instructions are useless, whereas in a real program at least one execution path have a certain number of useful instructions. Therefore, if the number of useful instructions in an execution path exceeds a threshold, we conclude the instruction sequence is a segment of a program.

Data Flow Anomaly The term data flow anomaly was originally used to analyze programs written in higher level languages in the software reliability and testing field [25, 26]. In this paper, we borrow this term and several other terms to analyze instruction sequences.

During a program execution, an instruction may impact a variable (register, memory location or stack) on three different ways: *define*, *reference*, and *undefine*. A variable is defined when it is set a value; it is referenced when its value is referred to; it is undefined when its value is not set or set by another undefined variable. Note that here the definition of undefined is different from that in a high level language. For example, in a C program, a local variable of a block becomes undefined when control leaves the block.

A data flow anomaly is caused by an improper sequence of actions performed on a variable. There are three data flow anomalies: *define-define*, *define-undefine*, and *undefine-reference* [26]. The define-define anomaly means that a variable was defined and is defined again, but it has never been referenced between these two actions. The undefine-reference anomaly indicates that a variable that was undefined receives a reference action. The define-undefine anomaly means that a variable was defined, and before it is used it is undefined. Figure 7 shows an example.

Detection of Data Flow Anomalies There are static [25] or dynamic [26] methods to detect data flow anomalies in the software reliability and testing field. Static methods are not suitable in our case due to its slow speed; dynamic methods are not suitable either due to the need for real execution of a program with some inputs. As such, we propose a new method called code abstraction, which does not require real execution of code. As a result of the code abstraction of an instruction, a variable

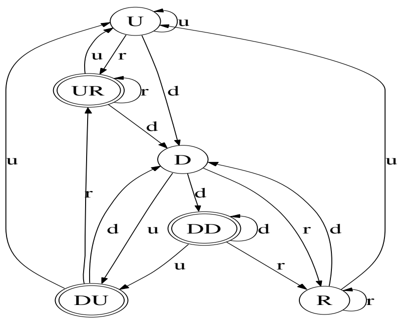


Figure 8: State diagram of a variable. State U: undefined, state D: defined but not referenced, state R: defined and referenced, state DD : abnormal state define-define, state UR: abnormal state undefine-reference and state DU: abnormal state define-undefine.

could be in one of the six possible states. The six possible states are state U: undefined; state D: defined but not referenced; state R: defined and referenced; state DD : abnormal state define-define; state UR: abnormal state undefine-reference; and state DU: abnormal state define-undefine. Figure 8 depicts the state diagram of these states. Each edge in this state diagram is associated with $d, r,$ or u, which represents "define", "reference", and "undefine", respectively.

We assume that a variable is in "undefined" state at the beginning of an execution path. Now we start to traverse this execution path. If the entry instruction of the execution path defines this variable, it will enter the state "defined". Then, it will enter another state according to the next instruction, as shown in Figure 8. Once the variable enters an abnormal state, a data flow anomaly is detected. We continue this traversal to the end of the execution path. This process enables us to find all the data flow anomalies in this execution path.

Pruning Useless Instructions Next we leverage the detected data flow anomalies to remove useless instructions. A *useless* instruction of an execution path is an instruction which does not affect the results of the execution path; otherwise, it is called *useful* instructions. We may find a useless instruction from a data flow anomaly. When there is an undefine-reference anomaly in an execution path, the instruction which causes the "reference" is a useless instruction. For instance, the instruction $K1$ in Figure 7, which causes undefine-reference anomaly, is a useless instruction. When there is a define-define or define-undefine anomaly, the instruction that caused the former "define" is also considered as a useless instruction. For instance, the instructions $I1$ and $J1$ in Figure 7 are useless instructions because they caused the former "define" in either the define-define or the define-undefine anomaly.

After pruning the useless instructions from an execution path, we will get a set of useful instructions. If the

Algorithm 2 check if the number of useful instructions in an execution path exceeds a threshold

```
Input: $entry$ instruction of an instruction sequence, EISG $G$
  $total \leftarrow 0$; $useless \leftarrow 0$ ; $stack \leftarrow$ empty
  initialize the $states$ of all variables to "undefined"
  push the entry $instruction$,$states$,$total$ and $useless$ to $stack$
  while $stack$ is not empty do
    pop the top item of $stack$ to $i$,$states$,$total$ and $useless$
    if $total - useless$ greater than a threshold then
      return true
    if $i$ is visited then
      continues
    mark $i$ visited
    $total \leftarrow total + 1$
    Abstractly execute instruction $i$ (change the $states$ of variables according to instruction $i$)
    if there is a define-define or define-undefine anomaly then
      $useless \leftarrow useless + 1$
    if there is a undefine-reference anomaly then
      $useless \leftarrow useless + 1$
    for each instruction $j$ directly following $i$ in the $G$ do
      push $j$, $states$ ,$total$ and $useless$ to $stack$
  return false
```

number of useful instructions in an execution path exceeds a threshold, we will conclude the instruction sequence is a segment of a program.

Algorithm 2 shows our algorithm to check if the number of useful instructions in an execution path exceeds a threshold. The algorithm involves a search over an EISG in which the nodes are visited in a specific order derived from a depth first search. The algorithm assumes that an EISG G and the entry instruction of the instruction sequence are given, and a push down stack is available for storage. During the search process, the visited node (instruction) is abstractly executed to update the states of variables, find data flow anomaly, and prune useless instructions in an execution path.

Handling Special Cases Next we discuss several special cases in the implementation of Scheme 2.

General purpose instruction The instructions in the IA32 instruction set can be roughly divided into four groups: general purpose instructions, floating point unit instructions, extension instructions, and system instructions. General purpose instructions perform basic data movement, arithmetic, logic, program flow, and string operation, which are commonly used by programmers to write applications and system software that run on IA-32 processors [3]. General purpose instructions are also the most often used instructions in malicious code. We believe that malicious codes must contain a certain number of general purpose instructions to achieve the attacking goals. Other types of instructions may be leveraged by an attacker to obfuscate his real-purpose code, e.g., used as garbage in garbage insertion. As such, we prune other groups of instructions as well.

Initial state of registers It is hard for attackers to know the run-time values of registers before malicious code is executed. That is, their values are unpredictable to attackers. Therefore, it is reasonable to assume that the initial states of all variables are "undefined" at the beginning of an execution path. The register "esp", however, is an exception since it is used to hold the stack pointer. Thus, we set register esp "defined" at the beginning of an execution path.

Indirect address An indirect address is an address that serves as a reference point instead of an address to the direct memory location. For example, in the instruction "move eax,[ebx+01e8]", register "ebx" may contain the actual address of the operand. However, it is difficult to know the run-time value of register "ebx". Thus, we always treat a memory location to which an indirect address points as state "defined" and hence no data flow anomaly will be generated. Indeed, this treatment successfully prevents an attacker from obfuscating his code using indirect addresses.

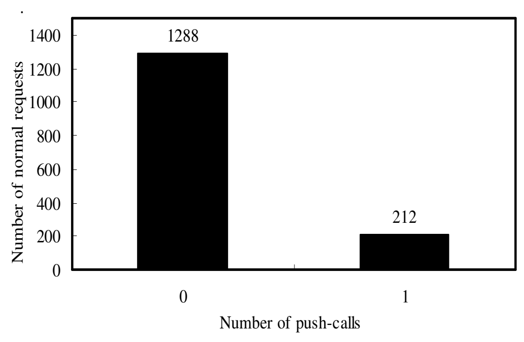
We will defer the discussion on the capability of Scheme 2 in defending against obfuscation until Section 7.

6 Experiments

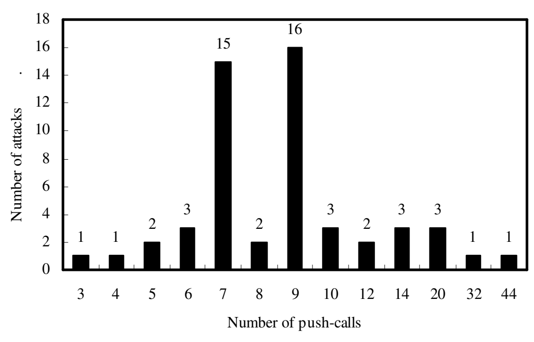
6.1 Parameter Tuning

Both Scheme 1 and Scheme 2 use a threshold value to determine if a request contains code or not. Clearly, it is critical to set the threshold values appropriately so as to minimize both detection false positive rate and false negative rate. To find out the appropriate thresholds, we tested both schemes of SigFree against 50 unencrypted attack requests generated by Metasploit framework, worm Slammer, CodeRed (CodeRed.a) and a CodeRed variation (CodeRed.c), and 1500 binary HTTP replies (52 encrypted data, 23 audio, 195 jpeg, 32 png, 1153 gif and 45 flash) intercepted on the network of College of Information Science and Technology. Note that we tested on HTTP replies rather than requests as normal data for parameter tuning, because HTTP replies include more diverse binaries (test over real traces of web requests is reported in Section 6.3). Also note that although worm Slammer attacks Microsoft SQL servers rather than web servers, it also exploits buffer overflow vulnerabilities.

Threshold of Push-calls for Scheme 1 Figure 9(a) shows that all instruction sequences distilled from a normal request contain at most one push-call code pattern.

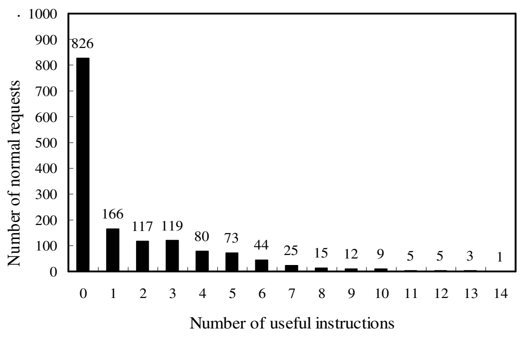


(a)

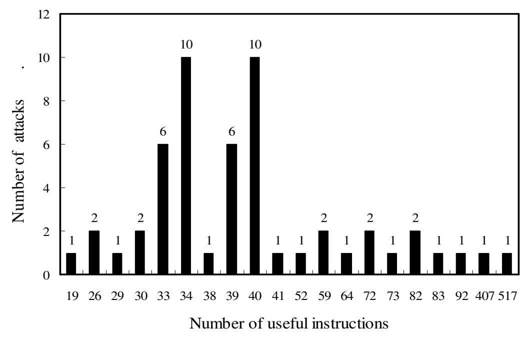


(b)

Figure 9: The number of push-calls in a request. (a) Normal requests. It shows that any instruction sequences of a normal request contain at most one push-call code pattern. (b) Attack requests. It shows that an attack request contains more than two external push-calls in one of its instruction sequences.



(a)



(b)

Figure 10: The number of useful instructions in a request. (a) Normal requests. It shows that no normal requests contain an instruction sequence which has over 14 useful instructions. (b) Attack requests. It shows that there exists an instruction sequence of an attack request which contain more than 18 useful instructions.

Figure 9(b) shows that for all the 53 buffer overflow attacks we tested, every attack request contains more than two push-calls in one of its instruction sequences. Therefore, by setting the threshold number of push-calls to 2, Scheme 1 can detect all the attacks used in our experiment.

Threshold of Useful Instructions for Scheme 2 Figure 10(a) shows that no normal requests contain an instruction sequence that has more than 14 useful instructions. Figure 10(b) shows that an attack request contains over 18 useful instructions in one of its instruction sequences. Therefore, by setting the threshold to a number between 15 and 17, Scheme 2 can detect all the attacks used in our test. The three attacks, which have the largest numbers of instructions (92, 407 and 517), are worm Slammer, CodeRed.a and CodeRed.c, respectively. This motivates us to investigate in our future work whether an exceptional large number of useful instructions indicates the occurrence of a worm.

6.2 Detection of Polymorphic Shellcode

We also tested SigFree on two well-known polymorphic engine, ADMmutate v0.84 [40] and CLET v1.0 [23]. Basically, ADMmutate obfuscates the shellcode of buffer overflow attacks in two steps. First, it encrypts the shellcode. Second, it obfuscates the decryption routine by substituting instructions and inserting junk instructions. In addition, ADMmutate replaces the No OPerations(NOP) instructions with other one-byte junk instructions to evade the detection of an IDS. This is because most buffer overflow attacks contain many NOP instructions to help locate shellcode, making them suspicious to an IDS.

CLET is a more powerful polymorphic engine compared with ADMmutate. It disguises its NOPs zone with 2,3 bytes instructions (not implemented yet in CLET v1.0), referred to as fake-NOPs, and generates a decipher routine with different operations at each time, which makes classical IDS pattern matching ineffective. Moreover, It uses spectrum analysis to defeat data mining methods.

Because there is no push-call pattern in the code, Scheme 1 cannot detect this type of attacks. However, Scheme 2 is still very robust to these obfuscation techniques. This is because although the original shellcode contains more useful instructions than the decryption routine has and it is also encrypted, Scheme 2 may still find enough number of useful instructions in the decryp-

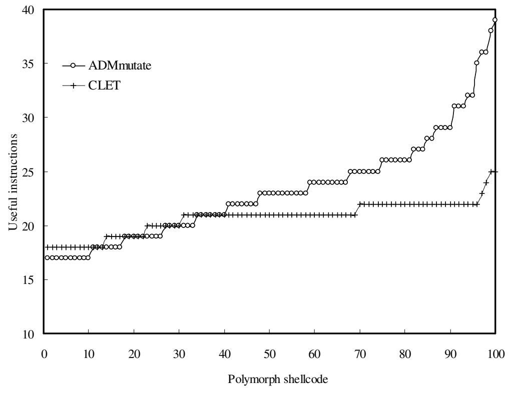


Figure 11: The number of useful instructions in all 200 polymorphic shellcodes. It shows that the least number of useful instructions in ADMmutate and CLET polymorphic shellcodes is 17.

tion routines.

We used each of ADMmutate and CLET to generate 100 polymorphic shellcodes, respectively. Then, we used Scheme 2 to detect the useful instructions in the code. Figure 11 shows the (sorted) numbers of useful instructions in 200 polymorphic shellcodes. We observed that the least number of useful instructions in these ADMmutate polymorphic shellcodes is 17, whereas the maximum number is 39; the least number of useful instructions in the CLET polymorphic shellcodes is 18, whereas the maximum number is 25. Therefore, using the same threshold value as before (i.e., between 15 and 17), we can detect all the 200 polymorphic shellcodes generated by ADMmutate and CLET.

6.3 Testing on Real Traces

We also tested SigFree over real traces. Due to privacy concerns, we were unable to deploy SigFree in a public web server to examine realtime web requests. To make our test as realistic as possible, we deployed a client-side proxy underneath a web browser. The proxy recorded a normal user's http requests during his/her daily Internet surfing. During a one-week period, more than ten of our lab members installed the proxy and helped collect totally 18,569 HTTP requests. The requests include manually typed urls, clicks through various web sites, searchings from search engines such as Google and Yahoo, secure logins to email servers and bank servers, and HTTPs requests. In this way, we believe our data set is diverse enough, not worse than that we might have got if we install SigFree in a single web server that provides only limited Internet services.

Our test based on the above real traces did not yield an alarm. This output is of no surprise because our normal web requests do not contain code.

6.4 Performance Evaluation

To evaluate the performance of SigFree, we implemented a proxy-based SigFree prototype using the C programming language in Win32 environment. SigFree was compiled with Borland C++ version 5.5.1 at optimization level O2. The prototype implementation was hosted in a Windows 2003 server with Intel Pentium 4, 3.2GHz CPU and 1G MB memory.

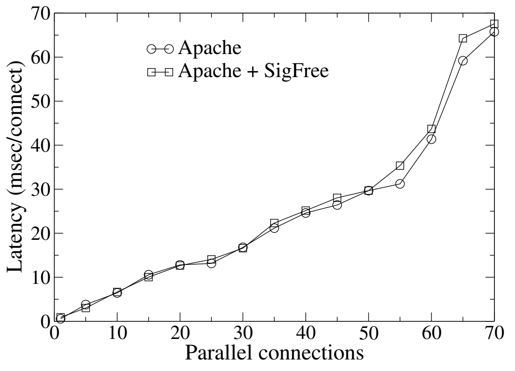
The proxy-based SigFree prototype accepts and analyzes all incoming requests from clients. The client testing traffics were generated by Jef Poskanzer's http_load program [3] from a Linux desktop PC with Intel Pentium 4 2.5GHz CPU connected to the Windows server via a 100 Mbps LAN switch. We modified the original http_load program so that clients can send code-injected data requests.

For the requests which SigFree identifies as normal, SigFree forwards them to the web server, Apache HTTP Server 2.0.54, hosted in a Linux server with dual Intel Xeon 1.8G CPUs. Clients send requests from a predefined URL list. The documents referred in the URL list are stored in the web server. In addition, the prototype implementation uses a time-to-live based cache to reduce redundant HTTP connections and data transfers.

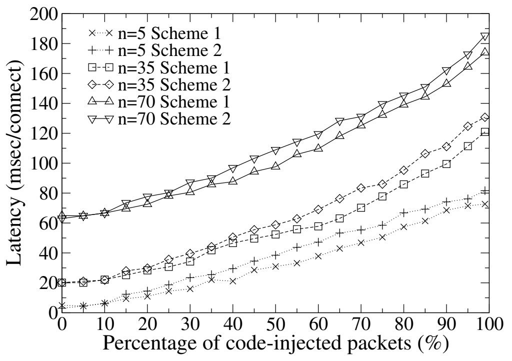
Rather than testing the absolute performance overhead of SigFree, we consider it more meaningful measuring the impact of SigFree on the normal web services. Hence, we measured the *average response latency* (which is also an indication of *throughput* although we did not directly measure throughput) of the connections by running http_load for 1000 fetches. Figure 12(a) shows that when there are no buffer overflow attacks, the average response time in the system with SigFree is only slightly higher than the system without SigFree. This indicates that, despite the connection and ASCII checking overheads, the proxy-based implementation does not affect the overall latency significantly.

Figure 12(b) shows the average latency of connections as a function of the percentage of attacking traffics. We used CodeRed as the attacking data. Only successful connections were used to calculate the average latency; that is, the latencies of attacking connections were not counted. This is because what we care is the impact of attack requests on normal requests. We observe that the average latency increases slightly worse than linear when the percentage of malicious attacks increases. Generally, Scheme 1 is about 20% faster than Scheme 2.

Overall, our experimental results from the prototype implementation show that SigFree has reasonably low performance overhead. Especially when the fraction of attack messages is small (say $< 10\%$), the additional latency caused by SigFree is almost negligible.



(a)



(b)

Figure 12: Performance impact of SigFree on Apache HTTP Server

7 Discussions

7.1 Robustness to Obfuscation

Most malware detection schemes include two-stage analysis. The first stage is disassembling binary code and the second stage is analyzing the disassembly results. There are obfuscation techniques to attack each stage [19, 38] and attackers may use them to evade detection. Table 1 shows that SigFree is robust to most of these obfuscation techniques.

Obfuscation in The First Stage *Junk byte insertion* is one of the simplest obfuscation against disassembly. Here junk bytes are inserted at locations that are not reachable at run-time. This insertion however can mislead a linear sweep algorithm, but can not mislead a recursive traversal algorithm [33], which our algorithm bases on.

Opaque predicates are used to transform unconditional jumps into conditional branches. Opaque predicates are predicates that are always evaluated to either true or false regardless of the inputs. This allows an obfuscator to insert junk bytes either at the jump target or in the place of the fall-through instruction. We note that opaque predicates may make SigFree mistakenly interpret junk byte as executable codes. However, this mistake will not cause SigFree to miss any real malicious instructions. Therefore, SigFree is also immune to obfuscation based on opaque predicates.

Obfuscation in The Second Stage Most of the second-stage obfuscation techniques obfuscate the behaviors of a program; however, the obfuscated programs still bear characteristics of programs. Since the purpose of SigFree is to differentiate executable codes and random binaries rather than benign and malicious executable codes, most of these obfuscation techniques are ineffective to SigFree. Obfuscation techniques such as instruction reordering, register renaming, garbage insertion and reordered memory accesses do not affect the number of calls or useful instructions which our schemes are based on. By exploiting instruction replacement and equivalent functionality, attacks may evade the detection of Scheme 1, but cannot evade the detection of Scheme 2.

Disassembly stage	Obfuscation	SigFree	
	Junk byte insertion	Yes	
	Opaque predict	Yes	
	Branch function	partial	
Analysis stage	Obfuscation	Scheme 1	Scheme 2
	Instruction reordering	Yes	Yes
	Register renaming	Yes	Yes
	Garbage insertion	Yes	Yes
	Instruction replacement	No	Yes
	Equivalent funcationality	No	Yes
	Reordered memory accesses	Yes	Yes

Table 1: SigFree is robust to most obfuscation

7.2 Limitations

SigFree also has several limitations. First, SigFree cannot fully handle the branch-function based obfuscation, as indicated in Table 1. Branch function is a function $f(x)$ that, whenever called from x, causes control to be transferred to the corresponding location $f(x)$. By replacing unconditional branches in a program with calls to the branch function, attackers can obscure the flow of control in the program. We note that there are no general solutions for handling branch function at the present state of the art.

With respect to SigFree, due to the obscurity of the flow of control, branch function may cause SigFree to break the executable codes into multiple instruction sequences. Nevertheless, it is still possible for SigFree to find this type of buffer overflow attacks as long as SigFree can still find enough push-calls or useful instructions in one of the distilled instruction sequences.

Second, the executable shellcodes could be written in alphanumeric form [45]. Such shellcodes will be treated as printable ASCII data and thus bypass our analyzer.

By turning off the ASCII filter, Scheme 2 can successfully detect alphanumeric shellcodes; however, it will increase unnecessary computational overhead. It therefore requires a slight tradeoff between tight security and system performance.

Finally, the current implementation of SigFree cannot detect malicious code which consists of fewer useful instructions than current threshold 15. Figure 13 shows a possible evasion which has only 7 useful instructions for a decryption routine. One solution to catch this evasion is to use a comprehensive score rather than the absolute number of useful instructions as the threshold. For example, we may give larger weights to instructions that are within a loop because most decryption routines contain loops. This approach, however, may introduce some false positives, which we will report in our future work.

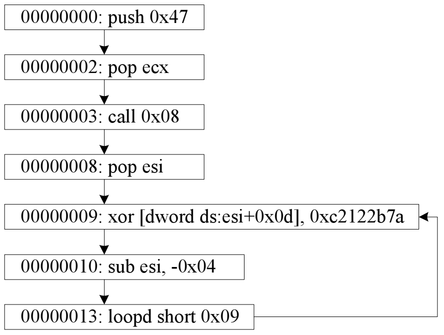

```
00000000: push 0x47
00000002: pop ecx
00000003: call 0x08
00000008: pop esi
00000009: xor [dword ds:esi+0x0d], 0xc2122b7a
00000010: sub esi, -0x04
00000013: loopd short 0x09
```

Figure 13: A decryption routine with 7 useful instructions. The first two instructions are used to set the initial value for loop counter ecx. The next two instructions are used to acquire the value of EIP (instruction pointer register). The last three instructions form the decryption loop.

7.3 Application-Specific Encryption Handling

The proxy-based SigFree could not handle encrypted or encoded data directly. A particular example is SSL-enabled web server. Enhancing security between web clients and web servers by encrypting HTTP messages, SSL also causes the difficulty for out-of-box malicious code detectors.

To support SSL functionality, an SSL proxy such as Stunnel [6] (Figure 14) may be deployed to securely tunnel the traffic between clients and web servers. In this case, we may simply install SigFree in the machine where the SSL proxy is located. It handles the web requests in cleartext that have been decrypted by the SSL proxy. On the other hand, in some web server applications, SSL is implemented as a server module (e.g., mod_ssl in Apache). In this case, SigFree will need to be implemented as a server module (though not shown in Figure 14), located between the SSL module and the WWW server. We notice that most popular web servers allow us to write a server module to process requests and specify the order of server modules. Detailed study will be reported in our future work.

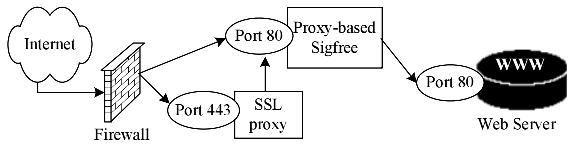


Figure 14: SigFree with a SSL proxy

7.4 Applicability

So far we only discussed using SigFree to protect web servers. It is worth mentioning that our tool is also widely applicable to many programs that are vulnerable to buffer overflow attacks. For example, the proxy-based SigFree may be used to protect all internet services which do not permit executable binaries to be carried in requests, e.g., database servers, email servers, name services, and so on. We will investigate the deployment issue in our future work.

In addition to protecting severs, SigFree can also provide file system real-time protection. Buffer overflow vulnerabilities have been found in some famous applications such as Adobe Acrobat and Adobe Reader [5], Microsoft JPEG Processing (GDI+) [1], and WinAmp [8]. This means that attackers may embed their malicious code in PDF, JPEG, or mp3-list files to launch buffer overflow attacks. In fact, a virus called Hesive [7] was disguised as a Microsoft Access file to exploit buffer overflow vulnerability of Microsoft's Jet Database Engine. Once opened in Access, infected .mdb files take advantage of the buffer overflow vulnerability to seize control of vulnerable machines. If mass-mailing worms exploit these kinds of vulnerabilities, they will become more fraudulent than before, because they may appear as pure data-file attachments. SigFree can be used alleviate these problems by checking those files and email attachments which should not include any code.

If the buffer being overflowed is inside a JPEG or GIF system, ASN.1 or base64 encoder, SigFree cannot be directly applied. Although SigFree can decode the protected file according to the protocols or applications it protects, more details need to be studied in the future.

8 Conclusion

We proposed SigFree, a realtime, signature free, out-of-the-box blocker that can filter code-injection buffer overflow attack messages, one of the most serious cyber security threats, to various Internet services. SigFree does not require any signatures, thus it can block new, unknown attacks. SigFree is immunized from most attack-side code obfuscation methods, good for economical Internet wide deployment with little maintenance cost and negligible throughput degradation, and can also handle encrypted SSL messages.

Acknowledgments We would like to thank our shepherd Marc Dacier and the anonymous reviewers for their valuable comments and suggestions. We are grateful to Yoon-Chan Jhi for helpful suggestions. We also thank the members of Penn State Cyber Security Lab for collecting real traces. The work of Xinran Wang and Sencun Zhu was supported in part by Army Research Office (W911NF-05-1-0270) and the National Science Foundation (CNS-0524156); the work of Chi-Chun Pan and Peng Liu was supported in part by NSF CT-3352241.

References

[1] Buffer overrun in jpeg processing (gdi+) could allow code execution (833987). http://www.microsoft.com/technet/security/bulletin/MS04-028.mspx

[2] Fnord snort preprocessor. http://www.cansecwest.com/spp_fnord.c.

[3] Intel ia-32 architecture software developer's manual volume 1: Basic architecture.

[4] Metasploit project. http://www.metasploit.com.

[5] Security advisory: Acrobat and adobe reader plug-in buffer overflow. http://www.adobe.com/support/techdocs/321644.html.

[6] Stunnel – universal ssl wrapper. http://www.stunnel.org.

[7] Symantec security response: backdoor.hesive. http://securityresponse.symantec.com/avcenter/venc/data/backdoor.hesive.html

[8] Winamp3 buffer overflow. http://www.securityspace.com/ smysecure/catid.html?id=11530.

[9] Pax documentation. http://pax.grsecurity.net/docs/pax.txt, November 2003.

[10] BARATLOO, A., SINGH, N., AND TSAI, T. Transparent run-time defense against stack smashing attacks. In *Proc. 2000 USENIX Technical Conference* (June 2000).

[11] BARRANTES, E., ACKLEY, D., PALMER, T., STEFANOVIC, D., AND ZOVI, D. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communications security* (October 2003).

[12] BERNERS-LEE, T., MASINTER, L., AND MCCAHILL, M. Uniform Resource Locators (URL). RFC 1738 (Proposed Standard). Updated by RFCs 1808, 2368, 2396, 3986.

[13] BHATKAR, S., SEKAR, R., AND DUVARNEY, D. C. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security* (2005).

[14] CHEN, H., DEAN, D., AND WAGNER, D. Model checking one million lines of c code. In *NDSS* (2004).

[15] CHINCHANI, R., AND BERG, E. V. D. A fast static analysis approach to detect exploit code inside network flows. In *RAID* (2005).

[16] CHRISTODORESCU, M., AND JHA, S. Static analysis of executables to detect malicious patterns. In *Proceedings of 12th USENIX Security Symposium* (August 2003).

[17] CHRISTODORESCU, M., JHA, S., SESHIA, S. A., SONG, D., AND BRYANT, R. E. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy, Oakland* (May 2005).

[18] CKER CHIUEH, T., AND HSU, F.-H. Rad: A compile-time solution to buffer overflow attacks. In *ICDCS* (2001).

[19] COLLBERG, C., THOMBORSON, C., AND LOW, D. A taxonomy of obfuscating transformations. Tech. Rep. 148, Department of Computer Science,University of Auckland, July 1997.

[20] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[21] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. Vigilante: End-to-end containment of internet worms. In *SOSP* (2005).

[22] COWAN, C., PU, C., MAIER, D., HINTON, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of 7th USENIX Security Conference* (January 1998).

[23] DETRISTAN, T., ULENSPIEGEL, T., MALCOM, Y., AND UNDERDUK, M. S. V. Polymorphic shellcode engine using spectrum analysis. http://www.phrack.org/show.php?p=61&a=9.

[24] EVANS, D., AND LAROCHELLE, D. Improving security using extensible lightweight static analysis. *IEEE Software 19*, 1 (2002).

[25] FOSDICK, L. D., AND OSTERWEIL, L. Data flow analysis in software reliability. *ACM Computing Surveys 8* (September 1976).

[26] HUANG, J. Detection of data flow anomaly through program instrumentation. *IEEE Transactions on Software Engineering 5*, 3 (May 1979).

[27] JUNG, J., PAXSON, V., BERGER, A., AND BALAKRISHNAN, H. Fast portscan detection using sequential hypothesis testing. In *Proc. IEEE Symposium on Security and Privacy* (2004).

[28] KC, G., KEROMYTIS, A., AND PREVELAKIS, V. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security* (October 2003).

[29] KIM, H.-A., AND KARP, B. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th Usenix Security Symposium* (August 2004).

[30] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. Secure execution via program shepherding. In *Proceedings of USENIX Security Symposium* (2002).

[31] KOLESNIKOV, O., AND LEE, W. Advanced polymorphic worms: Evading ids by blending in with normal traffic.

[32] KRUEGEL, C., KIRDA, E., MUTZ, D., ROBERTSON, W., AND VIGNA, G. Polymorphic worm detection using structural information of executables. In *RAID* (2005).

[33] KRUEGEL, C., ROBERTSON, W., VALEUR, F., AND VIGNA, G. Static disassembly of obfuscated binaries. In *Proceedings of USENIX Security 2004* (August 2004).

[34] KUPERMAN, B. A., BRODLEY, C. E., OZDOGANOGLU, H., VIJAYKUMAR, T. N., AND JALOTE, A. Detecting and prevention of stack buffer overflow attacks. *Communications of the ACM 48*, 11 (2005).

[35] LAKHOTIA, A., AND ERIC, U. Stack shape analysis to detect obfuscated calls in binaries. In *Proceedings of Fourth IEEE International Workshop on Source Code Analysis and Manipulation* (September 2004).

[36] LIANG, Z., AND SEKAR, R. Automatic generation of buffer overflow attack signatures: An approach based on program behavior models. In *Proceedings of the Annual Computer Security Applications Conference(ACSAC)* (2005).

[37] LIANG, Z., AND SEKAR, R. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proc. 12th ACM Conference on Computer and Communications Security* (2005).

[38] LINN, C., AND DEBRAY, S. Obfuscation of executable code to improve resistance to static disassembly. In *10th ACM Conference on Computer and Communications Security (CCS)* (October 2003).

[39] LOCASTO, M. E., WANG, K., KEROMYTIS, A. D., AND STOLFO, S. J. Flips: Hybrid adaptive intrusion prevention. In *RAID* (2005).

[40] MACAULAY, S. Admmutate: Polymorphic shellcode engine. http://www.ktwo.ca/security.html.

[41] MCGREGOR, J., KARIG, D., SHI, Z., AND LEE, R. A processor architecture defense against buffer overflow attacks. In *Proceedings of International Conference on Information Technology: Research and Education (ITRE)* (2003), pp. 243 – 250.

[42] NEWSOME, J., KARP, B., AND SONG, D. Polygraph: Automatic signature generation for polymorphic worms. In *IEEE Security and Privacy Symposium* (May 2005).

[43] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS* (2005).

[44] PANG, R., YEGNESWARAN, V., BARFORD, P., PAXSON, V., AND PETERSON, L. Characteristics of internet background radiation. In *Proc. ACM IMC* (2004).

[45] RIX. Writing ia32 alphanumeric shellcodes. http://www.phrack.org/show.php?p=57&a=15, 2001.

[46] SCHWARZ, B., DEBRAY, S. K., AND ANDREWS, G. R. Disassembly of executable code revisited. In *Proc. IEEE 2002 Working Conference on Reverse Engineering (WCRE)* (October 2002).

[47] SINGH, S., ESTAN, C., VARGHESE, G., AND SAVAGE, S. The earlybird system for real-time detection of unknown worms. Tech. rep., University of California at San Diego, 2003.

[48] S.KC, G., AND KEROMYTIS, A. D. e-nexsh: Achieving an effectively non-executable stack and heap via system-call policing. In *Proceedings of the Annual Computer Security Applications Conference(ACSAC)* (2005).

[49] SMIRNOV, A., AND CKER CHIUEH, T. Dira: Automatic detection, identification, and repair of control-hijacking attacks. In *NDSS* (2005).

[50] TOTH, T., AND KRUEGEL, C. Accurate buffer overflow detection via abstract payload execution. In *RAID* (2002).

[51] WAGNER, D., FOSTER, J. S., BREWER, E. A., AND AIKEN, A. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium* (February 2000).

[52] WANG, H. J., GUO, C., SIMON, D. R., AND ZUGENMAIER, A. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the ACM SIGCOMM Conference* (August 2004).

[53] WANG, K., , CRETU, G., AND STOLFO, S. J. Anomalous payload-based worm detection and signature generation. In *RAID* (2005).

[54] WANG, K., AND STOLFO, S. J. Anomalous payload-based network instrusion detection. In *RAID* (2004).

[55] XU, J., NING, P., KIL, C., ZHAI, Y., AND BOOKHOLT, C. Automatic diagnosis and response to memory corruption vulnerabilities. In *Proc. 12th ACM Conference on Computer and Communications Security* (2005).

Notes

[1]An attack may direct execution control to existing system code or change the values of certain function arguments.

[2]http://www.research.ibm.com/trl/projects/security/ssp/

[3]http://www.acme.com/software/http_load/

Polymorphic Blending Attacks

Prahlad Fogla   Monirul Sharif   Roberto Perdisci   Oleg Kolesnikov   Wenke Lee

College of Computing, Georgia Institute of Technology
801 Atlantic Drive, Atlanta, Georgia 30332
{prahlad, msharif, rperdisc, ok, wenke}@cc.gatech.edu

Abstract

A very effective means to evade signature-based intrusion detection systems (IDS) is to employ polymorphic techniques to generate attack instances that do not share a fixed signature. Anomaly-based intrusion detection systems provide good defense because existing polymorphic techniques can make the attack instances look different from each other, but cannot make them look like normal. In this paper we introduce a new class of polymorphic attacks, called *polymorphic blending attacks*, that can effectively evade byte frequency-based network anomaly IDS by carefully matching the statistics of the mutated attack instances to the normal profiles. The proposed polymorphic blending attacks can be viewed as a subclass of the *mimicry* attacks. We take a systematic approach to the problem and formally describe the algorithms and steps required to carry out such attacks. We not only show that such attacks are feasible but also analyze the hardness of evasion under different circumstances. We present detailed techniques using `PAYL`, a byte frequency-based anomaly IDS, as a case study and demonstrate that these attacks are indeed feasible. We also provide some insight into possible countermeasures that can be used as defense.

1 Introduction

In the continuing arms race in computer and network security, a common trend is that attackers are employing polymorphic techniques. Toolkits such as ADMmutate [17], PHATBOT [10], and CLET [5] are available for novices to generate polymorphic attacks. The purpose of using polymorphism is to evade detection by an intrusion detection system (IDS). Every instance of a polymorphic attack *looks* different and yet carries out the same malicious activities. For example, the payload of each instance of a polymorphic worm can have different byte contents. It follows that signature-based (misuse) IDS may not *reliably* detect a polymorphic attack because it may not have a fixed or predictable signature, or because the invariant parts of the attack may not be sufficient to construct a signature that produces very few false positives. On the other hand, each instance of a polymorphic attack needs to contain exploit code that is typically not used in normal activities. Thus, each instance looks different from normal. Existing polymorphic techniques [28] focus on making the attack instances look different from each other, and not much on making them look like normal. This means that network payload anomaly detection systems can provide a good defense against the current generation of polymorphic attacks. However, if a polymorphic attack can blend in with (or look like) normal traffic, it can successfully evade an anomaly-based IDS that relies solely on payload statistics.

In this paper, we show that it is possible to evade network anomaly IDS based on payload statistics using a class of polymorphism that we call polymorphic blending. A polymorphic blending attack is a polymorphic attack that also has the ability to evade a payload statistics-based anomaly IDS. In addition to making all the mutated attack instances different, an attacker (or the attack code) attempts to make them appear normal by transforming each instance in such a way that its payload characteristics (e.g., the byte frequency distribution) fit the normal profile used by the anomaly IDS. Since polymorphic blending attacks try to evade the IDS by making the attacks look like normal, they can be viewed as a subclass of the *mimicry* attacks [29, 33].

This paper makes several contributions. We study the class of polymorphic blending attacks against byte frequency-based network anomaly IDS, which was introduced by Kolesnikov et al. in [12]. We present the general techniques and design considerations for such attacks. We provide rationales of why these attacks are practical and show that network anomaly IDS based on payload statistics do not guarantee adequate protection against sophisticated attacks.

Using 1-gram and 2-gram PAYL [35, 36] as a case study, we take a systematic approach to the problem and describe the necessary steps required to carry out an effective attack. Our work provides insight into not only how such an attack can be performed, but also how hard it is to launch these attacks under different circumstances. We analyze the amount of learning required for the attacker and the time and space complexity required for blending. We use a real attack vector [8] to implement a polymorphic blending attack and provide experimental evidence that our attack can effectively evade detection. We also discuss possible countermeasures that a defender (e.g., IDS designer or operator) can take to decrease the likelihood that a polymorphic blending attack will succeed.

Organization of the paper The rest of the paper is organized as follows. We discuss related work in polymorphic attacks and detection in Section 2. In Section 3, we introduce polymorphic blending attacks and discuss the general techniques and design issues of polymorphic blending attacks. We present our case study in Section 4 and conclude the paper in Section 5.

2 Related Work

Transforming attack packets to avoid detection is a common practice among attackers. Attackers can exploit the ambiguities present in the traffic stream to transform an attack instance to another so that an IDS is not able to recognize the attack pattern. IP and TCP transformations ([11, 22]) techniques are used to evade NIDS that analyzes TCP/IP headers. Vigna et al. [31] discussed multiple network, application and exploit layer (shellcode polymorphism) mutation mechanisms. A formal model to combine multiple transformations was presented by Rubin et al. [24]. Multiple tools such as Fragroute [26], Whisker [23], and AGENT [24] are available that can perform attack mutation.

Code polymorphism has been used extensively by virus writers to write polymorphic viruses. Mistfall, tPE, EXPO, and DINA [28, 37] are some of the polymorphic engines used by virus writers. Worm writers have also started using polymorphic engines. ADMmutate [17], PHATBOT [10], and JempiScodes [25] are some of the polymorphic shellcode generators commonly used to write polymorphic worms. Garbage and NOP insertions, register shuffling, equivalent code substitution, and encryption/decryption are some of the common techniques used to write polymorphic shellcodes.

Quite a few approaches have been proposed to detect polymorphic attacks. In [30], Toth et al. proposed a technique to locate the presence of executable shellcode inside the payload. They used abstract execution of network flows to find the *MEL* (Maximum Executable Length) of the payload. The flow is marked suspicious if its *MEL* is above certain length. Chinchani et al. [2] performed fast static analysis to check if a network flow contains exploit code. STRIDE [1] focuses on detecting polymorphic sleds used by buffer overflow attacks. In [14], Kruegel et al. used structural analysis of binary code to find similarities between different worm instances. Using a graph coloring technique on a worm's control flow graph, this approach is able to accurately model the structure of the worm. Given a set of suspicious flows, Polygraph [20] generates a set of disjoint invariant substrings that are present in multiple suspicious flows. These substrings can then be used as a signature to detect worm instances. In a recent work, Perdisci et al. [21] proposed an attack on Polygraph [20] where noise is injected into the dataset of suspicious flows so that Polygraph is not able to generate a reliable signature for the worm. Shield [34] uses transport layer filters to block the traffic that exploits a known vulnerability. Filters are exploit-independent, and vulnerabilities are described as a partial state machines of the vulnerable application. In [3], Christodorescu et al. proposed an instruction semantics based worm detection technique. The proposed approach can detect code polymorphism that uses instruction reordering, register shuffling, and garbage insertions. It is worth noting that unless the attacker combines the polymorphic blending attack proposed in this paper with other evasion techniques, the approaches cited above [1, 2, 3, 14, 20, 30, 34] may be able to detect the attack. We further discuss possible countermeasures against the polymorphic blending attack in Section 4.7.

A number of attacks aimed at evading Host-based anomaly IDS have been developed. Wagner et al. [33] and Tan et al. [29] presented mimicry attacks against the stide model [9] developed by Forrest et al. The main idea behind these mimicry attacks was to inject dummy system calls into an attack sequence to make the final system call sequence look similar to the normal system call sequence. As a defense against mimicry attacks as well as other *impossible path* attacks [7, 32], more advanced detection approaches (e.g., [6, 7]) were proposed, which use call stack information along with the system call sequences. Recently, a more sophisticated mimicry attack was proposed by Kruegel et al. [13], which can evade most system call based anomaly IDS.

Several application payload-based anomaly IDS [15, 18, 19] have been proposed which monitor the payload of a packet for anomalies. In [16], Kruegel et al. proposed four different models, namely, length, character distribution, probabilistic grammar, and token finder, for the detection of HTTP attacks. PAYL, proposed by Wang and Stolfo [35], records the average frequency of occurrences of each byte in the payload of a normal packet. A separate profile is created for each port and packet length. In their recent work [36], the authors suggested an improved version of PAYL that computes several profiles for each port. At the end of the training, clustering is performed to reduce the number of profiles. They proposed that instead of byte frequency, one can also use an *n*-gram model in a similar fashion. One main drawback of the system is that they do not consider an advanced attacker, who may know the IDS running at the target and actively try to evade it. In this paper we provide strong evidence that such byte frequency based anomaly IDS are open to attacks and may be easily evaded.

CLET [5], an advanced polymorphic engine, comes closest to our polymorphic blending attack. It performs spectrum analysis to evade IDS that use data mining methods for worm detection. Given an attack payload, CLET adds padding bytes in a separate *cramming bytes* zone (of a given length) to try and make the byte frequency distribution of the attack close to the normal traffic. However, the encoded shellcode (using *XOR*) in CLET may still deviate significantly from the normal distribution and the obtained polymorphic attack may be detected by the IDS. A preliminary work by Kolesnikov et al. [12] introduced and cursorily explored polymorphic blending attacks. In this paper we present a systematic approach for evading byte frequency-based network anomaly IDS, and provide detailed analysis of the design, complexity and possible countermeasures for the polymorphic blending attacks. We also show that our polymorphic blending technique is much more effective than CLET in evading byte frequency-based anomaly IDS.

3 Blending Attacks

3.1 Polymorphism

A *polymorphic attack* is an attack that is able to change its appearance with every instance. Thus, there may be no fixed or predictable signature for the attack. As a result, it may evade detection because most current intrusion detection systems and anti-virus systems are signature-based. Exploit mutation and shellcode polymorphism are two common ways to generate polymorphic attacks. In general, there are three components in a polymorphic attack:

1. Attack Vector: an attack vector is used for exploiting the vulnerability of the target host. Certain parts of the attack vector can be modified to create mutated but still valid exploits. There might still be certain parts, called the invariant, of the attack vector that have to be present in every mutant for the attack to work. If the attack invariant is very small and exists in the normal traffic, then an IDS may not be able to use it as a signature because it will result in a high number of false positives.

2. Attack Body: the code that performs the intended malicious actions after the vulnerability is exploited. Common techniques to achieve attack body (shellcode) polymorphism include register shuffling, equivalent instruction substitution, instruction reordering, garbage insertions, and encryption. Different keys can be used in encryption for different instances of the attack to ensure that the byte sequence is different every time.

3. Polymorphic Decryptor: this section contains the part of the code that decrypts the shellcode. It decrypts the encrypted attack body and transfers control to it. Polymorphism of the decryptor can be achieved using various code obfuscation techniques.

Detection of Polymorphic Attacks All attack instances contain exploit code and/or input data that are typically not used in normal activities. For example, an attack instance, especially its decryptor and encrypted shellcode, may contain characters that have very low probability of appearing in a normal packet. Thus, an anomaly-based IDS can detect the polymorphic attack instances by recognizing their deviation from the normal profile. For example, Wang et al. [35, 36] showed that the byte frequency distribution of an (polymorphic) attack is quite different from that of normal traffic, and can thus be used by the anomaly-based IDS `PAYL` to detect simple polymorphic attacks. However, detection of a sophisticated polymorphic attack is much more challenging.

3.2 Blending Attacks

Clearly, if a polymorphic attack can "blend in" with (or "look" like) normal, it can evade detection by an anomaly-based IDS. Normal traffic contains a lot of syntactic and semantic information, but only a very small amount of such information can be used by a *high speed* network-based anomaly IDS. This is due to fundamental difficulties in modeling complex systems and performance overhead concerns in real-time monitoring. The network traffic profile used by *high speed* anomaly IDS, e.g., `PAYL`, typically includes simple statistics such as maximum or average size and rate of packets, frequency distribution of bytes in packets, and range of tokens at different offsets.

Given the incompleteness and the imprecision of the normal profiles based on simple traffic statistics, it is quite feasible to launch what we call *polymorphic blending attacks*. The main idea is that, when generating a polymorphic attack instance, care can be taken so that its payload characteristics, as measured by the anomaly IDS, will match the normal profile. For example, in order to evade detection by `PAYL` [35, 36], the polymorphic attack instance can carefully choose the characters used in encryption and pad the attack payload with a chosen set of characters, so that the resulting byte frequency of the attack instance closely matches the normal profiles and thus will be considered normal by PAYL.

3.2.1 A Realistic Attack Scenario

Before presenting the general strategies and techniques used in polymorphic blending attacks, we present an attack scenario and argue that such attacks are realistic. Figure 1 shows the attack scenario that is the basis of our case study. There are a few assumptions behind this scenario:

- The adversary has already compromised a host X inside a network A which communicates with the target host Y inside network B. Network A and host X may lack sufficient security so that the attack can penetrate without getting detected, or the adversary may collude with an insider.
- The adversary has knowledge of the IDS (IDS_B) that monitors the victim host network. This might be possible using a variety of approaches, e.g., social engineering (e.g., company sales or purchase data), fingerprinting, or trial-and-error. We argue that one *cannot* assume that the IDS deployment is a secret, and security by obscurity is a very weak position. We assume IDS_B is a payload *statistics* based system (e.g., PAYL). Since the adversary knows the learning algorithm being used by IDS_B, given some packet data from X to Y, the adversary will be able to generate its *own* version of the *statistical* normal profile used by IDS_B.
- A typical anomaly IDS has a threshold setting that can be adjusted to obtain a desired false positive rate. We assume that the adversary does not know the exact value of the threshold used by IDS_B, but has an estimation of the generally acceptable false positive and false negative rates. With this knowledge, the adversary can estimate the error threshold when crafting a new attack instance to match the IDS profile.

We now explain the attack scenario. Once the adversary has control of host X, it observes the normal traffic going from X to Y. The adversary estimates a normal profile for this traffic using the same modeling technique that IDS_B uses. We call this an *artificial profile*. With it, the adversary creates a mutated instance of itself in such a way that the statistics of the mutated instance match the artificial profile. When IDS_B analyzes these mutated attack packets, it is unable to discern them from normal traffic because the artificial profile can be very close to the actual profile in use by IDS_B. Thus, the attack successfully infiltrates the network B and compromises host Y.

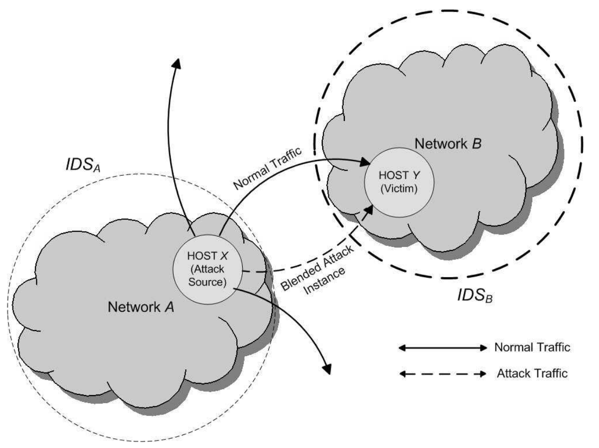


Figure 1: Attack Scenario of Polymorphic Blending Attack

3.2.2 Desirable Properties of Polymorphic Blending Attacks

Clearly, the key for a polymorphic blending attack to succeed in evading an IDS is to be able to learn an artificial profile that is very close to the actual normal profile used by the IDS, and create polymorphic instances that match the artificial profile. There are other desirable properties. First, the blending process (e.g., with encoding and padding) should not result in an abnormally large attack size. Otherwise, a simple detection heuristic will be to monitor the network flow size. Second, although we do not put any constraint on the resources available to the adversary, the polymorphic blending process should be economical in terms of time and space. Otherwise, it will not only slow down the attack, but also increase the chance of detection by the local IDS (e.g., IDS_A or host-based IDS.) More formally, given a description of the algorithm that the IDS uses to learn and match the normal profile and an attack instance, the time (and space) complexity of the algorithm used to apply polymorphic blending to the attack instance should be a small degree polynomial with respect to the initial attack size. Algorithms that require exponential time and space may not be practical. Since the learning time should be small, the blending algorithm should not require to collect a lot of normal packets to learn the normal statistics.

3.3 Steps of Polymorphic Blending Attacks

The polymorphic blending attack has three basic steps: (1) learn the IDS normal profile; (2) encrypt the attack body; (3) and generate a polymorphic decryptor.

3.3.1 Learning The IDS Normal Profile

The task at hand for the adversary is to observe the normal traffic going from a host, say X, to another host in the target network, say Y, and generate a normal profile close to the one used by the IDS at the target network, say IDS_B, using the same algorithm used by the IDS.

A simple method to get the normal data is by sniffing the network traffic going from network A to host Y. This can be easily accomplished in a bus network. In a switched environment, it may be harder to obtain such data. Since the adversary knows the type of service running at the target host, he can simply generate normal request packets and learn the artificial profile using these packets.

In theory, even if the adversary learns a profile from just a single normal packet, and then mutates an attack instance so that it matches the statistics of the normal packet perfectly, the resulting polymorphic blended attack packet should not be flagged as an anomaly by IDS_B, provided the normal packet does not result in a false positive in the first place. On the other hand, it is beneficial to generate an artificial profile that is as close to the normal profile used by IDS_B as possible, so that if a polymorphic blended attack packet matches the artificial profile closely it has a high chance of evading IDS_B. In general, if more normal packets are captured and used by the adversary, she will be able to learn an artificial normal profile that is closer to the normal profile used by IDS_B.

3.3.2 Attack Body Encryption

After learning the normal profile, the adversary creates a new attack instance and encrypts (and blends) it to match the normal profile. A straightforward byte substitution scheme followed by padding can be used for encryption. The main idea here is that every character in the attack body can be substituted by a character(s) observed from the normal traffic using a substitution table. The encrypted attack body can then be padded with some more garbage normal data so that the polymorphic blended attack packet can match the normal profile even better. To keep the padding (and hence the packet size) minimal, the substituted attack body should already match the normal profile closely. We can use this design criterion to produce a suitable substitution table.

To ensure that the substitution algorithm is reversible (for decrypting and running the attack code), a one-to-one or one-to-many mapping can be used. A single-byte substitution is preferred over multi-byte substitution because multi-byte substitution will inflate the size of the attack body after substitution. An obvious requirement of such encryption scheme is that the encrypted attack body should contain characters from only the normal traffic. Although this may be hard for a general encryption technique (because the output typically looks random), it is an easy requirement for a simple byte substitution scheme. However, finding an optimal substitution table that requires minimal padding is a complex problem. In Section 4, we show that for certain cases this is a very hard problem. We can instead use a greedy method to find an acceptable substitution table. The main idea is to first sort the statistical features in the descending order of the frequency for both the attack body and normal traffic. Then, for each unassigned entry with the highest frequency in the attack body, we simply map it to an available (not yet mapped) normal entry with the highest frequency. This procedure is repeated until all entries in the attack body are mapped. The feature mapping can be translated to a character mapping and a substitution table can be created for encryption and decryption purposes.

3.3.3 Polymorphic Decryptor

A decryptor first removes all the extra padding from the encrypted attack body and then uses a reverse substitution table (or decoding table) to decrypt the attack body to produce the original attack code (shellcode).

The decryptor is not encrypted but can be mutated using multiple iterations of shellcode polymorphism processing (e.g., mapping an instruction to an equivalent one randomly chosen from a set of candidates). To reverse the substitution done during blending, the decryptor needs to look up a decoding table that contains the required reverse mappings. The decoding table for one-to-one mapping can be stored in an array where the i-th entry of the array represents the normal character used to substitute attack character i. Such an decoding table contains only normal characters. Unused entries in the table can be used for padding. On the other hand, storage of decoding tables for one-to-many mapping or variable-length mapping is complicated and typically requires larger space.

3.4 Attack Design Issues

3.4.1 Incorporating Attack Vector and Polymorphic Decryptor in Blending

We discussed in Section 3.3.2 that the encryption of the attack body is guided by the need to make the attack packet match the normal statistical profile (or more precisely, the learned artificial profile).

The attack vector, decryptor, and substitution table are not encrypted. Their addition to the attack packet payload alters the packet statistics. The new statistics may deviate significantly from the normal profile. In such a case, we must find a new substitution table in order to match the whole attack packet to the normal profile. First, we take the normal profile and subtract the frequencies of characters in the attack vector, decryptor, and existing substitution table. Next, we find a new substitution table using the adjusted normal profile. If the statistics of the new substitution table is not significantly different from the old substitution table, we use the new substitution table for encryption. Otherwise we repeat the above steps.

3.4.2 Packet Length based IDS Profile

If IDS_B has different profiles for packets of different lengths, as in the case of PAYL, the substitution phase and padding phase need to use the normal profile corresponding to the final attack packet size. A target length greater than the length of the original attack packet (before polymorphic blending) is chosen at first. The encryption step is then applied and the packet is padded to the target length. If the statistics of the resulting attack packet is not very close to the normal profile, a different target length is chosen and the above process is repeated. Another strategy is to divide the attack body into multiple small packets and perform the polymorphic blending process for all of them separately.

4 Evaluation and Results

To demonstrate that polymorphic blending attacks are feasible and practical, we show how an attack can use polymorphic blending to evade the anomaly IDS PAYL.

In this section, we first describe the polymorphic blending techniques to evade PAYL. Then we report the results of the experiments we ran to evaluate the evasion capabilities of the polymorphic blending attacks.

In our evaluation, we first established a baseline performance by sending polymorphic instances (generated using the CLET polymorphic engine) of the attack to PAYL and verified that all of the instances were detected by the IDS as anomalies. Then, without changing the configuration of PAYL, we used our polymorphic blending techniques to generate attack instances to see how well they can evade the IDS.

4.1 PAYL Anomaly IDS as A Case Study

PAYL has been shown to be effective in detecting polymorphic attacks and worms [35, 36]. For this reason we used PAYL in our case study. We used the 2-gram version in addition to the 1-gram version to evaluate how polymorphic blending attack is affected when an IDS uses a more comprehensive model.

PAYL uses n-gram analysis by recording the frequency distribution of n-grams in the payload of a packet. A sliding window of width n is used to record the number of occurrences of all the n-grams present in the payload. A separate model is generated for each packet length. These models are clustered together at the end of the training to reduce the number of models. Furthermore, the length of a packet is also monitored for anomalies. Thus a packet with an unseen or very low frequency length is flagged as an anomaly. $\{f(x_i), \sigma(x_i)\}$ represents the PAYL model of normal traffic, where x_i is the ith gram, which is a character in 1-gram PAYL, and a tuple in 2-gram PAYL. $f(x_i)$ is the average relative frequency of x_i in the normal traffic, and $\sigma(x_i)$ is the standard deviation of x_i in the normal traffic. The anomaly score as calculated by PAYL is shown in Equation 1.

$$score(P) = \sum_i (\mathring{f}(x_i) - f(x_i))/(\sigma(x_i) + \alpha) \qquad (1)$$

Here, P is the monitored packet, $\mathring{f}(x_i)$ is the relative frequency of the ith gram x_i in P, and α is a smoothing factor used to prevent division by zero. For convenience we will use the term frequency to denote relative frequency.

We evaluated our polymorphic blending attack with the first version of PAYL as described in [35]. Wang *et al.* [36] proposed some improvements on PAYL in their recent version. We believe that our attack still works for this new version of PAYL. The main improvement of the new version is to use multiple centroids for a given packet length, so that a low false positive rate can be achieved using a relatively low anomaly threshold. In this case, our polymorphic blending attack has to use the same learning algorithm as the new version of PAYL. Furthermore, more normal traffic needs to be used to learn an artificial profile that is close to the actual normal profile. Thus, the effect is that our attack may take a little more time. The new version also matches ingress *suspicious* traffic with egress *suspicious* traffic to find worms. This feature does not have any effect on our attack because the attack instances blend in with normal.

4.2 Evading 1-gram

To evade 1-gram PAYL, the frequency of each character in the attack packet should be close to the average frequency recorded during the learning phase. We substitute the characters in the attack packet with the characters seen in the normal traffic, and apply sufficient amount of padding so that the 1-gram frequencies of the resulting packet match the normal profile very closely. We first present analytical results on the amount of padding required to match the substituted attack body with the normal profile perfectly. Then we present a substitution algorithm that uses the padding criteria to minimize the amount of required padding.

In the following sections, we assume that the normal frequency $f(x)$ has already been adjusted for the attack vector, the decryptor, and the decoding table (as discussed in Section 3.4.1, these parts need to be accounted for when computing the frequencies of characters to find a suitable substitution).

4.2.1 Padding

Let $\hat{w}$ and $\acute{w}$ be the substituted attack body before and after padding, respectively. Let n be the number of distinct characters in the normal traffic. $\|s\|$ denotes the length of a string s, and λ_i denotes the number of occurrences of the normal character x_i in the padding section of the blending packet. Then,

$$\|\acute{w}\| = \|\hat{w}\| + \sum_{i=1}^{n} \lambda_i \tag{2}$$

Suppose the relative frequency of character x_i in the normal traffic and the substituted attack body is $f(x_i)$ and $\hat{f}(x_i)$, respectively. Since the final desired frequency of x_i is $f(x_i)$, the number of occurrences of x_i in the blending packet should be $\|\acute{w}\| f(x_i)$. Thus, λ_i can be defined using the following equation:

$$\lambda_i = \|\acute{w}\| f(x_i) - \|\hat{w}\|\hat{f}(x_i), \quad 1 \le i \le n \tag{3}$$

Equation 3 can be re-written as,

$$\|\acute{w}\| = \frac{\lambda_i + \|\hat{w}\|\hat{f}(x_i)}{f(x_i)}, \quad 1 \le i \le n \tag{4}$$

Since $f(x)$ and $\hat{f}(x)$ are relative frequency distributions, $\sum_i f(x_i) = \sum_i \hat{f}(x_i) = 1$. Unless they are identical, there exists some character x_i for which $\hat{f}(x_i) > f(x_i)$. The character x_i is perhaps "overused" in the substituted attack body. It is trivial to see that we need to pad all the characters except the one that is most overused. Let x_k be the character that has highest overuse and δ be the degree of overuse. That is,

$$\delta = \delta_k = max_i\{\delta_i\}, \text{ where } \delta_i = \frac{\hat{f}(x_i)}{f(x_i)}, \quad 1 \le i \le n \tag{5}$$

Since no padding is required for character x_k, $\lambda_k = 0$. Putting this value in Equation (4) we get:

$$\|\acute{w}\| = \frac{0 + \|\hat{w}\|\hat{f}(x_k)}{f(x_k)} = \delta\|\hat{w}\| \tag{6}$$

The amount of padding required for each character x_i can be calculated by substituting the value of $\|\acute{w}\|$ in Equation (3):

$$\lambda_i = \|\hat{w}\|(\delta f(x_i) - \hat{f}(x_i)) \tag{7}$$

Thus, using the padding defined by the above equation, we can match the final attack packet perfectly to the normal frequency $f(x)$. Furthermore, the amount of padding required by the above equation is the minimum amount that is needed to match the normal profile exactly. Please refer to Appendix 6.1 for the proof.

4.2.2 Substitution

The analysis in Section 4.2.1 shows that the amount of padding can be minimized by minimizing δ, which is $max(\frac{\hat{f}(x_i)}{f(x_i)})$. This in turn means that the objective of the substitution process is to minimize the resulting δ. There are two possible cases for substitution. The first is when the number of distinct characters present in the attack body (m) is less than or equal to the number of distinct characters present in the normal traffic (n), i.e. $m \le n$. In this case we can perform single-byte encoding, either one-to-one or one-to-many. If $m > n$, we need to use multi-byte encoding.

Case: $m \le n$ We suggest a greedy algorithm to generate a one-to-many mapping from the attack characters to the normal characters that provides an acceptable solution and is computationally efficient. Our algorithm tries to minimize the ratio δ locally for each substitution assignment.

Let x_i represents a normal character and y_j represent an attack character. Let $f(x_i)$ be the frequency of character x_i in normal traffic and $g(y_j)$ be the frequency of character y_j in the attack body. Let $S(y_j)$ be the set of normal characters to which y_j is mapped. Let $\hat{tf}(y_j) = \Sigma_{x_i \in S(y_j)} f(x_i)$. The probability that y_j is substituted by $x_i, x_i \in S(y_j)$, during substitution is $\frac{f(x_i)}{\hat{tf}(y_j)}$. Thus, the number of occurrences of x_i in the substituted attack body is $\frac{f(x_i) \times g(y_j)}{\hat{tf}(y_j)}$. We then have $\delta_i = \frac{(f(x_i) \times g(y_j)/\hat{tf}(y_j))}{f(x_i)} = \frac{g(y_j)}{\hat{tf}(y_j)}$. Our greedy algorithm tries to minimize this ratio δ_i locally. The substitution algorithm is as follows.

Sort the normal character frequency $f(x)$ and the attack character frequency $g(y)$ in descending order. For the first m characters, map y_i to x_i and set $S(y_i) = \{x_i\}$ and $\hat{tf}(y_i) = f(x_i), \forall 1 \le i \le m$. For the $(m+1)th$ normal character, x_{m+1}, find an attack character (y_j) with maximum ratio of $\frac{g(y_j)}{\hat{tf}(y_j)}$. Assign x_{m+1} to y_j and set $S(y_j) = \{x_{m+1}\} \cup S(y_j)$ and $\hat{tf}(y_j) = \hat{tf}(y_j) + f(x_{m+1})$. This is performed for each of the remaining characters until we reach the end of the frequency list $f(x)$. While substituting alphabet y_j in the attack body, we choose a character x_i from the set $S(y_j)$ with probability $\frac{f(x_i)}{\hat{tf}(y_j)}$.

Consider an example where $f(a, b, c) = \{0.3, 0.4, 0.3\}$, attack body $w = qpqppqpq$, and $g(p, q) = \{0.5, 0.5\}$. According to the above algorithm, initially, b and a are assigned to p and q respectively. At this point, ratio $\frac{g(p)}{\hat{tf}(p)} = 1.25$ and $\frac{g(q)}{\hat{tf}(q)} = 1.66$. So we assign c to q. Thus, p will be substituted by b and q will be substituted by a with probability 0.5 and by c with probability 0.5. Thus, the attack after substitution can be

$\hat{w} = cbabbcba$.

In our experiments, we used a simple one-to-one mapping where characters with the highest frequencies in the attack packet are mapped to characters with the highest frequencies in normal traffic. This simple mapping is shown to be sufficient for the blending purpose.

Case: $m > n$ We suggest a heuristic based on Huffman encoding scheme to obtain a small attack size after encoding. Given the frequency distribution of the characters in the attack body being encoded, Huffman encoding provides a minimum length packet after encoding. The weights of the nodes in Huffman tree is the sum of the relative frequencies of all its descendant leaf nodes. The weight of a leaf node is the frequency of a given character in the attack body. Every edge in the tree is assigned to a character from the normal profile. In the original Huffman coding the edges of the Huffman tree are labeled randomly. Random labeling of the edges may give us a very large value of δ. We developed a heuristic to assign labels to edges of Huffman tree to find a mapping that gives us a very small δ. Before stating the heuristic, we present the problem of optimally assigning the labels to the edges in Huffman tree:

Given a Huffman tree, assign labels $l(v) \in N$ to the vertices v in the tree, such that after substitution, $\delta = max(\frac{\hat{f}(x)}{f(x)}), \forall x \in N$, is minimum. The constraint on the label $l(v)$ is that if $parent(v_1) = parent(v_2)$, then $l(v_1) \neq l(v_2)$.

We propose a greedy algorithm to find an approximate solution for the above problem. First sort the vertices in descending order of their weight and initialize the capacity of each character $cap(x_i) = f(x_i), \forall x_i \in N$. Then starting from the leftmost unlabeled vertex v_j, find a character x_i with the maximum $cap(x_i)$ and that is not assigned to any of the direct siblings of v_j. Assign x_i to v_j and reduce the capacity of x_i by the weight of the vertex. Repeat until all the vertices are assigned. The labels generated by the above algorithm are used for the substitution process. An example is explained in Figure 2.

4.3 Evading 2-gram

The 1-gram PAYL model assumes that the bytes occurring in the stream are independent. It does not try to capture any information of byte sequencing of the normal traffic. The 2-gram model on the other hand can capture some byte sequencing information. It records the frequencies of all the 2-grams present in the normal traffic. It is easy to see that by matching 2-grams we are inherently performing 1-gram matching as well.

For 2-gram, the polymorphic blending process needs to match the frequencies of not only all the characters but also all the tuples. Similar to 1-gram substitution, one can either use single-byte encoding or multi-byte encoding for substitution. For single-byte encoding, the goal is to find a one-to-one or one-to-many mapping that ensures that all the tuples in the substituted attack body are also present in normal profile. In Appendix 6.2, we show that this is NP-complete for the general case by reducing the well known sub-graph isomorphism problem [4] to the mapping problem. Unlike single-byte encoding, it is possible for an attacker to find a multi-byte encoding scheme that produces only valid 2-grams. Here, we present a viable multi-byte encoding scheme.

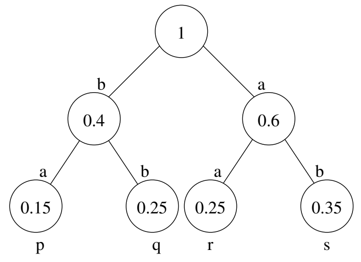


Figure 2: 1-gram multibyte encoding. The frequency of the normal character is $f(a,b) = \{0.5, 0.5\}$. Sorted weights of the nodes are $\{0.6, 0.4, 0.35, 0.25, 0.25, 0.15\}$. Using the proposed algorithm we get $S : \{p,q,r,s\} \mapsto \{ba, bb, aa, ab\}$

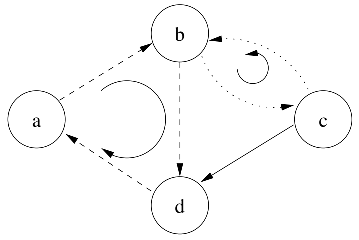


Figure 3: 2-gram multibyte encoding. $e_0 = da,\ e_1 = bc$. $w = 01101010$. $\hat{w} = bdabcbcbdabcbdabcbda$

4.3.1 Multi-byte Encoding

A 2-gram normal profile can be viewed as a Moore machine (FSM) which has a state for each character in N. Every state is a start state and end state. A transition from state v_1 to state v_2 exists if and only if 2-gram v_1v_2 exists in normal profile. This FSM represents the language accepted by the IDS with given 2-gram profile. Strings generated by the FSM contain only normal 2-grams. Characters in an attack body can be mapped to paths in this FSM. For example, suppose the state machine has two cycles reachable from each other. e_1 and e_2 be two edges such that e_1 is present only in the first cycle and e_2 is present only in the second cycle. Given a bit representation of the attack body, we can encode 0 using e_0 and 1 using e_1. We can generate any bit string represented using these two tuples interleaved by other non-informative characters present in the cycles and in the paths between two cycles. Figure 3 shows an example of such an encoding scheme. Such an encoded attack string will have a very large size. We use it to show the existence of an encoding scheme that is able to match the normal 2-grams. We can generate a more efficient encoding scheme by using the entropy measure of transitions at each state. The complete details of such an encoding scheme are not addressed in this paper. The authors suggest readers to refer to coding theory for more on entropy based encoding.

4.3.2 Approximate Single-Byte Encoding

As discussed above, the problem of finding a single-byte substitution is hard for 2-gram. On the other hand, multi-byte encoding may increase the size of the attack packets considerably. We can use a simple approximation algorithm to find a good one-to-one substitution. The algorithm performs single byte substitution in such a way that tuples with high frequencies in the attack packet are greedily matched with tuples with high frequencies in normal traffic.

The details of the algorithm are as follows. First, sort the normal tuple frequencies $f(x_{i,j})$ and the attack tuple frequencies $g(y_{i,j})$ in descending order. Initially, all tuples in the list $f(x_{i,j})$ are marked $unused$ and the substitution table is cleared. The frequency list $g(y)$ is traversed from the top. For every tuple $y_{i,j}$ in the sorted attack tuple list, the list $f(x)$ is traversed from the beginning to find an $unmarked$ tuple $x_{i',j'}$ so that substituting y_i with $x_{i'}$ and y_j with $x_{j'}$ does not violate any mappings that were already made. The tuple $x_{i',j'}$ is $marked$ and the substitution table is updated. The above algorithm is fast and provides consistent reversible matching. The algorithm does not guarantee to provide the best substitution, i.e., the closest distance to the target frequency distribution.

4.3.3 Padding

We introduce an efficient padding algorithm that does not provide minimal padding but tries to match the target distribution in a greedy manner. Let $d_f(x_{i,j})$ be the difference between the frequency of tuple $x_{i,j}$ in the normal profile and the substituted attack body. Find a tuple $x_{k,l}$ from the list of normal tuples that starts with the last padded character (x_k) and that has the highest $d_f(x_{k,m}), \forall 1 \leq m \leq 256$. The second character of the tuple, x_l, is padded to the end of the packet and $d_f(x_{k,l})$ is reduced. This step is repeated until the blending attack size reaches a desired length.

4.4 Complexity of Blending Attacks

We now summarize the methods provided above and analyze the hardness of a polymorphic blending attack while keeping the design goals (Section 3.2.2) in mind. For 1-gram blending, although finding a substitution that minimizes the padding seems to be a hard-problem and may take exponential time, we have proposed greedy algorithms that find a good substitution that require small amount of padding to perfectly match the normal byte frequency. For 2-gram blending, finding a single-byte substitution that ensures only normal tuples after substitution is shown to be NP-hard (see the proof in Appendix 6.2). An approximation algorithm can be used to efficiently compute a substitution that may introduce a few invalid tuples. A multibyte encoding scheme can achieve a very good match with no invalid tuples at the expense of very large attack sizes. An attacker has to therefore consider several trade-offs between the degree of matching, attack size, and time complexity to mount successful blending attacks.

4.5 Experiment Setup

4.5.1 Attack Vector

We chose an attack that targets a vulnerability in Windows Media Services (MS03-022). The attack vector we selected exploits a problem with the logging ISAPI extension that handles incoming client requests. It is based on the implementation by `firew0rker` [8]. The size of the attack vector is 99 bytes and is required to be present at the start of the `HTTP` request. The attack needs to send approximately 10KB of data to cause the buffer overflow and compromise the system. Our attack body opens a `TCP` client socket to an IP address and sends system registry files. The size of the unencrypted attack body is 558 bytes and contains 109 different characters. During the blending process, we divided our attack into several packets. If our final blending attack after padding does not add up to 10KB, we just send some normal packets as a part of the attack to cause the buffer overflow. The decryptor was divided into multiple sections and distributed among different packets. The attack body was divided among all the attack packets.

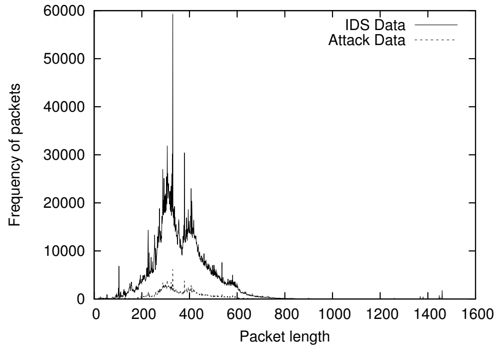


Figure 4: Packet length distribution

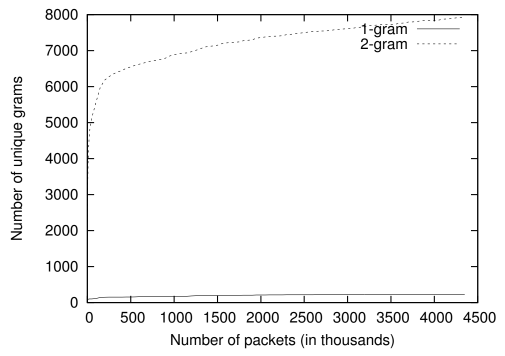


Figure 5: Observed Unique 1-grams and 2-grams

4.5.2 Dataset

Data Type	Feature	Packet length		
		418	730	1460
IDS Training	Num. of Pkts	16,490	540	1,781
	One Grams	106	90	128
	Two Grams	4,325	3,791	3,903
Attack Training	Num. of Pkts	2,168	82	249
	One Grams	89	86	86
	Two Grams	2,847	2,012	2,196

Table 1: HTTP Traffic dataset

We collected around 15 days of HTTP traffic coming to our department's network in November 2004. We used several IDSs, including *Snort*, to verify that this data contains no known attack. We removed all the packets with no TCP payload. We used the data of the first 14 days (4,356,565 packets, 1.9GB) for IDS training to obtain the IDS normal profiles. A separate profile was created for each TCP payload length (or simply packet length). The full payload section of each packet was used to compute the profiles. The last day of the HTTP traffic was made available to the attacker to learn the artificial profile. We also used cross-validation, i.e., randomly picking one of the 15 days for attack training and the rest for IDS training, to verify the results of our experiments.

The packet length distributions in the IDS training dataset and the attack training dataset are shown in Figure 4. Among this packet lengths, we chose three different lengths to implement the blending attack, namely 418, 730 and 1460. These packets lengths are large enough to accommodate the attack data into a small number of packets. These lengths also occurred frequently in the training dataset. A separate artificial profile was created for each packet length using the attack training data of the same packet length. Thus, we generated three 1-gram models and three 2-gram models for different packet lengths. Table 1 shows the details of the datasets used for the evaluation. The numbers of unique 1-grams and 2-grams in the data are also shown in the table.

4.6 Evaluation

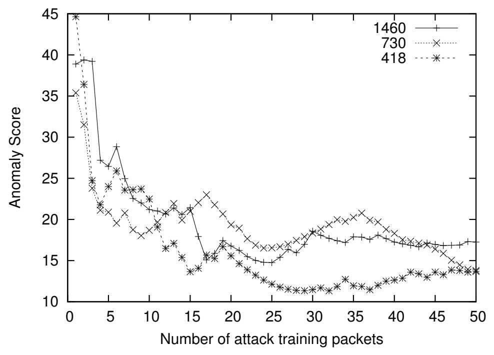
Training time of 1-gram and 2-gram PAYL: We performed experiments on the training time required to learn the profiles used by PAYL. Figure 5 shows the numbers of unique 1-grams and 2-grams observed in HTTP traffic stream. Since the numbers of observed 1-gram and 2-gram continue to increase as new packets arrive in the stream, the training of profiles for 1-gram and 2-gram takes a long time to converge. We trained our IDS model using all of the available IDS training data.

Traditional polymorphic attacks: To the best of our knowledge, CLET [5] is the only publicly available tool that implements evasion techniques against byte frequency-based anomaly IDS. For this reason we used CLET as our baseline. As mentioned in Section 2, given an attack CLET adds padding bytes in the payload to make the byte frequency distribution of the attack close to the normal traffic. However, CLET does not apply any byte substitution technique (see Section 4.2.2). Further, CLET does not address the evasion of 2-gram PAYL explicitly. We also generated polymorphic attacks using other well known tools (e.g., ADMutate [17]), and verified that they are less effective than CLET in evading PAYL.

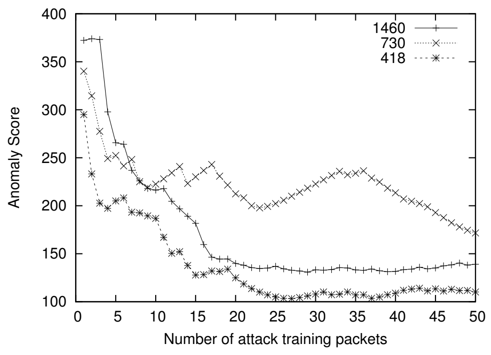
We generated multiple polymorphic instances of our attack body using CLET and tested them against PAYL. Each attack instance contained one or more attack packets of given length. Different amount of bytes were crammed (padded) to obtain the desired attack size. Attack training data was used to generate spectral files used for cramming by the CLET engine. A polymorphic attack instance will evade an IDS model if and only if all the attack packets corresponding to the attack instance are able to evade the IDS. Thus, the anomaly score of an attack instance was calculated as the highest of all the anomaly scores (Equation 1) obtained by the attack packets corresponding to the attack instance. Table 2 shows the anomaly threshold setting of different PAYL models that result in the detection of all the attack instances. The anomaly thresholds were calculated as the minimum anomaly score over all the attack instances. Using the given thresholds, both 1-gram and 2-gram PAYL were successful in detecting all the instances of the attack. Having established this "baseline" performance, we would like to show that our blending attacks can evade PAYL even if a lower threshold is used.

Packet Length	1-gram	2-gram
418	872	1,399
730	652	1,313
1460	355	977

Table 2: IDS anomaly threshold setting that detects all the polymorphic attacks sent by the CLET engine



(a) 1-gram



(b) 2-gram

Figure 6: Anomaly score of Artificial Profile

Packet Length	1-gram	2-gram
418	8	20
730	8	18
1460	14	40

Table 3: Number of packets required for the convergence of attacker's training

4.6.1 Artificial Profile

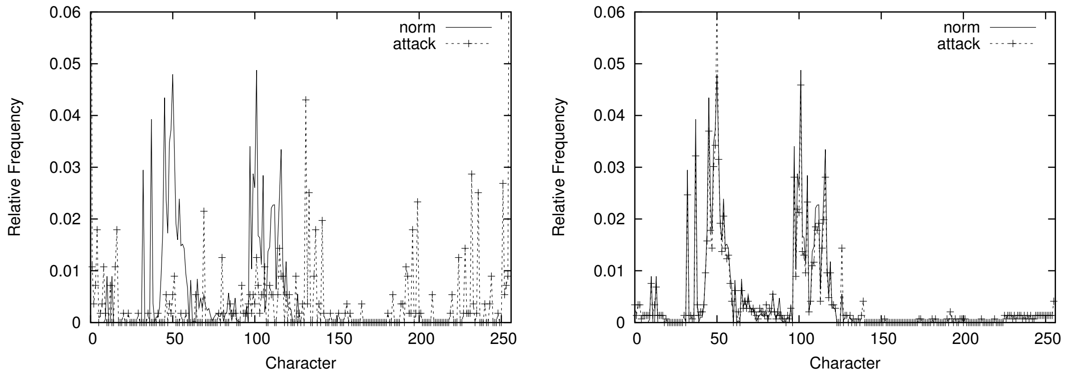
We used a simple convergence technique, similar to PAYL, to stop the training of the artificial profile. At every certain interval (convergence check interval) we check if the $Manhattan$ [1] distance between the artificial profiles at the last interval and the current interval is smaller than a certain threshold (convergence threshold). It stops training if the distance is smaller than the threshold. We set the convergence threshold ($= 0.05$) to be the same as the original implementation of PAYL. The artificial profile does not have to become very stable or match the normal profile perfectly because some deviation from the normal profile can be tolerated. To reduce the training time we set the convergence check interval to 2 packets. Thus, if we see two consecutive packets of a given length that are close to the learned profile, we stop training. Table 3 shows the number of packets required to converge the artificial profile of different packet lengths. As expected, the artificial profile converges very fast. The 1-gram profile converges faster than the 2-gram profile for the same packet length. We show that a small number of packets are enough to create an effective polymorphic blending attack. In practice, the attacker can use more learning data to create a better profile.

Figure 6 shows the anomaly score of the artificial normal profile, as calculated by the IDS normal profile, versus the number of attack training packets used to learn the artificial profile. As the number of attack training packets increases, the anomaly score of artificial normal profile decreases, which means that the artificial profile trained using more packets is a better estimation of the PAYL normal profile. The score needs to be less than the anomaly threshold of PAYL for the blending attack packets to have a realistic chance of evading PAYL. For all attack training sizes shown in Figure 6, the score is well under the threshold (Table 2) used to configure PAYL to detect all the traditional (without blending) polymorphic attack instances.

4.6.2 Blending Attacks for 1-gram and 2-gram PAYL

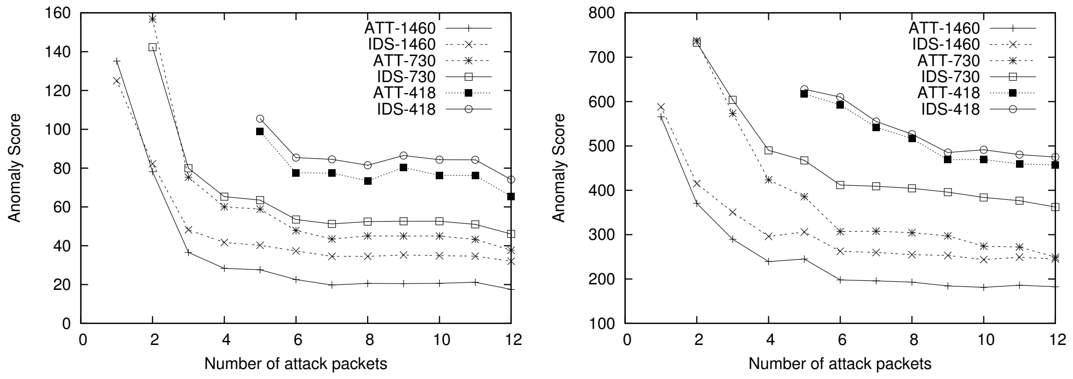
For each packet length, we generated both the 1-gram and 2-gram PAYL normal profiles using the entire IDS training dataset (i.e., the first 14 days of HTTP traffic). For each packet length, the 1-gram and 2-gram artificial normal models were learned using a fraction of the attack training dataset. The learning stops at the point the models converge, as shown in Table 3.

We used the one-to-one single-byte substitution technique discussed in Section 4.2.2 for constructing the blending attack against 1-gram PAYL, and the single byte encoding scheme discussed in Section 4.3.2 for the blending attack against 2-gram PAYL. Two sets of blending experiments were performed. In the first set of experiments, the substituted attack body was divided into multiple packets and each packet was padded separately to match the normal profile. A single decoding table is required to decode the whole attack flow. In the second set of experiments, the attack body was first divided into a given number of packets. Each of the attack body sections were substituted using one-to-one single byte substitution and then padded to match the normal frequency. Individually substituting the attack body for each packet allowed us to match the statistical profile of the substituted attack body closer to the normal profile. But it requires a separate decoding table for each packet, thus reducing the padding space considerably. For convenience, we call the first set of experiments



(a) Original attack packet (b) 1-gram Blending Packet for packet length 1460

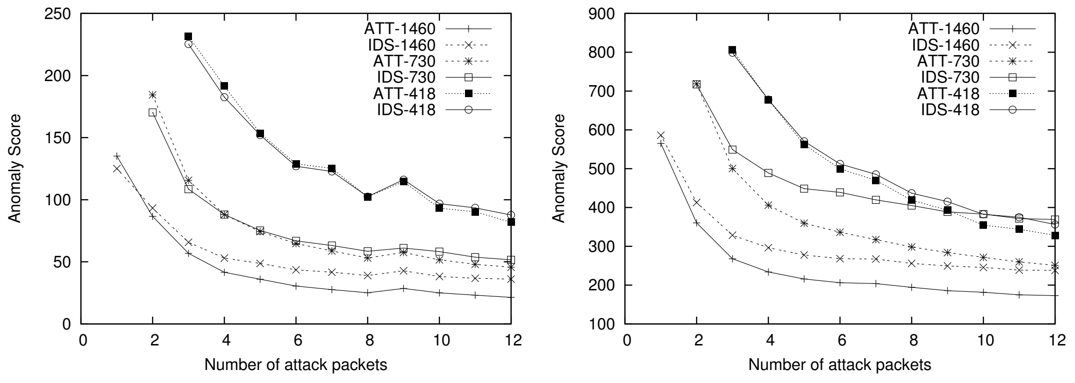
Figure 7: Comparison of frequency distribution of normal profile and attack packet



(a) 1-gram (b) 2-gram

Figure 8: Anomaly score of the blending attack packets (with local substitution) for artificial profile and IDS profile

global substitution, and the second *local substitution*. If $m > n$ for any of the above experiments, we simply substituted the low frequency attack characters using non-existing characters in the normal. This increased the error in blending attack but reduced the complexity of the blending attack algorithm. Figure 7 shows the comparison of the frequency distribution of different characters present in the HTTP traffic. The byte frequency distribution of the original attack instance is very different from the normal profile because the normal data has mainly printable ASCII characters whereas the attack payload has many characters that are unprintable. Thus, this was easily detected by both 1-gram and 2-gram IDS models. The attack was substituted and padded to obtain a single packet of length 1460. As shown in Figure 7(b), the frequency distribution of attack payload after substitution and padding becomes almost identical to the `PAYL` normal profile. This demonstrates the effectiveness of our polymorphic blending techniques. We studied how dividing an attack instance into several packets and blending them separately help match the attack packets with the artificial profile and evade `PAYL`. The experiments were performed with the number of attack packets ranging from 1 to 12. We checked the anomaly score of each attack packet as calculated by both the artificial profile and the IDS profile. Similar to the anomaly score of attack instances generated by CLET, the anomaly score of a blending attack instance was calculated as the highest of all the scores obtained by the attack packets corresponding to the blending attack instance. Figure 8 and Figure 9 show the anomaly scores of blending attacks with local substitution and global substitution, respectively. For each attack flow, we show the score of the packet with the highest score. It is evident that if the attack is divided into more packets, it matches the profile more closely. The reason is that if the attack body is divided into multiple fragments, for each packet there is more padding space available to match the profile. Also, local substitution works better than global substitution scheme for all cases except for 2-gram blending for packet length 418. Since our substitution table contains only normal 1-grams but may contain foreign 2-grams, a large substitution table may produce a large error for the 2-gram model. Considering



(a) 1-gram (b) 2-gram

Figure 9: Anomaly score of the blending attack packets (with global substitution) for artificial profile and IDS profile

that small packets have small padding space to reduce the error caused by the substitution table, having an individual substitution table in each packet can cause large error.

Although the score of the blending attack as calculated by the IDS model is greater than the score calculated by the artificial normal profile, it is still much lower than the anomaly threshold set for the detection of traditional polymorphic attacks.

Thus, our experiment clearly shows that unlike traditional polymorphic attacks, our blending attack is very effective in evading 1-gram and 2-gram PAYL for all the packet lengths and number of attack packets.

4.6.3 IDS False Positive Rate And Its Impact on Blending Attacks

We also studied the effect of false positive rates on the detection of blending attacks. Anomaly threshold for a given false positive rate (fp) is set such that only fp fraction of normal data has anomaly score higher than the anomaly threshold. The anomaly thresholds for different false positive rates are shown in Table 4. The number of attack packets required to evade the IDS successfully for a given threshold is shown in the parenthesis. As we increase the false positive rate, we need to divide the attack into more packets to keep the score below the anomaly threshold. Thus, keeping a high false positive rate may increase the size of the blending attack. From the table we can infer that even if the IDS keeps its false positive rate high to detect more attacks, blending attack can still easily evade it using an attack size as small as 3,650, i.e. five packets of length 730.

Since 2-gram PAYL records some sequence information along with byte frequencies, it seems to be a good representation of normal traffic. In our experiments we found that 2-gram PAYL consistently produces higher anomaly score than 1-gram PAYL for all attack packet lengths. But at the same time, the 2-gram IDS needs to set very high anomaly thresholds to avoid high false positive rates. Thus, in practice, the 2-gram PAYL is actually only marginally more effective than the 1-gram version in detecting attacks.

Blending attacks can be successfully launched on both 1-gram and 2-gram models. Larger packet lengths are more suitable for blending attacks. With few exceptions, the local substitution scheme works better than the global substitution scheme. The 2-gram model provides only marginal advantage over the 1-gram model in detecting blending attacks but requires huge space to store the model. Thus, the 2-gram model may not be a better choice over the 1-gram model.

4.7 Countermeasures

The experimental results reported above show that the statistical models used by PAYL are not sufficiently accurate to detect deliberate evasion attempts. We believe this problem is common to other network anomaly IDS that use traffic statistics [15, 18]. By following the ideas presented in this paper, it may be fairly easy to devise different blending algorithms in order to evade other network anomaly IDSs that rely solely on some form of packet statistics. The reason is that traffic statistics used by such network-based anomaly IDS do not provide a comprehensive representation of normal traffic. Application syntax and semantics related information cannot be modeled accurately using simple statistics of network packets. On the other hand, some of the IDS introduced in Section 2, e.g., [1, 2, 30], use syntax and semantics related information and could be used to detect the polymorphic blending attack. Nevertheless, modeling application syntax and semantic information is in general more expensive than measuring simple traffic statistics. Thus the trade-off between detection accuracy, hardness of evasion and operational speed has to be considered. A key direction to explore is to develop a more efficient semantic-based IDS that can be deployed on high-speed

False Positive	418		730		1460	
	1-gram	2-gram	1-gram	2-gram	1-gram	2-gram
0.1	61.07 (17,-)	373.4 (-,12)	63.70 (5,7)	467.6 (5,5)	74.50 (3,3)	447.7 (2,2)
0.01	78.61 (12,15)	456.9 (22,8)	143.6 (2,3)	625.5 (3,3)	81.98 (3,3)	531.0 (2,2)
0.001	125.5 (5,7)	561.8 (7,6)	164.6 (2,3)	670.5 (3,3)	239.2 (1,1)	931.9 (1,1)
0.0001	166.8 (5,5)	582.6 (7,5)	244.5 (2,2)	805.0 (2,2)	243.4 (1,1)	935.0 (1,1)

Table 4: Anomaly thresholds for different false positive rates in IDS models. Bracketed entries are the the numbers of packets required to evade the IDS using the local and global substitution scheme, respectively.

networks.

Another defense approach is to use multiple IDS models that use independent features. Such a collective set of models may be a better representation of the normal traffic. In such a case, a polymorphic blending attack will need to evade all (or the majority) of the models.

One reason blending attacks work is that the attacker has the complete knowledge of the IDS model being used. This gives the attacker an enormous advantage. A possible countermeasure is to introduce randomness [27] in the IDS model. Consider a model constructed by measuring the occurrence frequency of pairs of non-consecutive bytes that are separated by ν number of bytes. For example, given a payload containing the sequence of byte values $\{b_1, b_2, \cdots, b_l\}$, the IDS could measure the occurrence frequency of the pair of byte values $(b_i, b_{i+\nu+1}), \forall i = 0, \cdots, (l - \nu - 1)$, where l is the payload length. We call this a 2_ν-gram model. For $\nu = 0$, the 2_ν-gram model is the same as the 2-gram `PAYL` model. If the IDS chooses ν at random during the training phase, this makes the blending attack more difficult given that the attacker needs to guess the value of ν before applying the blending algorithm (note that ν is chosen at random before the model is created and is fixed for each packet. Therefore, the 2_ν-gram model is as complex as the 2-gram model used by `PAYL`). Furthermore, the IDS could construct m different models, each of them having a different randomly chosen ν_k, with $k = 1, \cdots, m$, and combine their output in order to obtain a more accurate decision about the packets. In this case the attacker needs to guess m values for the parameter ν and needs to devise a blending algorithm that "satisfies" all the m different models at the same time. This means that even if the attacker knows exactly how the IDS performs the training and test phases, it is much more difficult to evade it.

Preliminary experimental results show that if ν is small with respect to the payload size, the 2_ν-gram model is able to capture a sufficient amount of structural information that allows to construct an accurate IDS model. Further, the combination of different 2_ν-gram models appears to be a promising technique. However, the complexity of the detection system grows linearly with m. A thorough analysis of this modeling technique is beyond the scope of this paper and will be the subject of our future work.

While countermeasures may make evasion harder to succeed, they typically require more resources and can be more complex in design and implementation. It may also produce higher error rates if the IDS uses too many features such that its models "overfit" the data. In short, trade-offs between "hardness of evasion" and other performance measures need to be carefully considered.

5 Conclusion

In this paper, we presented a new class of attacks called polymorphic blending attacks. Existing polymorphic techniques can be used for evading signature-based IDS because the attack instances do not share a consistent signature. But anomaly IDS can detect these attack instances because the polymorphism techniques fail to mask their statistical anomalies. Our proposed attack overcomes this very shortcoming. The idea is to first learn the normal profiles used by the IDS, and then, while creating a polymorphic instance of an attack, make sure that its statistics match the normal profiles.

We described the basic steps and general techniques that can be used to devise polymorphic blending attacks. We presented a case study using the anomaly IDS `PAYL` to demonstrate that these attacks are practical and feasible. Our experiments showed that polymorphic blending attacks can evade `PAYL` while traditional polymorphic attacks cannot. We also showed that an attacker does not need a large number of packets to learn the normal profile and blend in successfully. The results with 2-gram `PAYL` suggested that simply using more complex features or models do not always provide a good defense against these polymorphic blending attacks. We discussed some possible defenses against polymorphic blending attacks.

Acknowledgments

This work is supported in part by NSF grant CCR-0133629 and Office of Naval Research grant N000140410735. The contents of this work are solely the responsibility of the authors and do not necessarily represent the official views of NSF and the U.S. Navy. The authors would like to thank the anonymous reviewers for helpful comments and the shepherd of this paper Professor Fabian Monrose at The Johns Hopkins University for very valuable suggestions.

References

[1] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. D. Anagnostakis. Stride: Polymorphic sled detection through instruction sequence analysis. *In Proceedings of the 20th IFIP International Information Security Conference (IFIP/SEC 2005)*, 2005.

[2] R. Chinchani and E.V.D. Berg. A fast static analysis approach to detect exploit code inside network flows. *In Recent Advances in Intrusion Detection*, 2005.

[3] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-aware malware detection. *In Proceeding of the IEEE Security and Privacy Conference*, 2005.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to algorithms. *The MIT Press*, 1990.

[5] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. Underduk. Polymorphic shellcode engine using spectrum analysis. *Phrack Issue 0x3d*, 2003.

[6] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizing sensitivity in static analysis for intrusion detection. *In Proceedings the IEEE Symposium on Security and Privacy*, 2004.

[7] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. *In Proceedings of the IEEE Security and Privacy Conference*, 2003.

[8] Firew0rker. Windows media services remote command execution exploit. *http://www.k-otik.com/exploits/07.01.nsiilog-titbit.cpp.php*, 2003.

[9] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff. A sense of self for unix processes. *In Proceedings of the IEEE Symposium on Security and Privacy*, 1996.

[10] Threat Intelligence Group. Phatbot trojan analysis. *http://www.lurhq.com/phatbot.html*.

[11] M. Handley and V. Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. *In 10th USENIX Security Symposium*, 2001.

[12] O. M. Kolesnikov, D. Dagon, and W. Lee. Advanced polymorphic worms: Evading IDS by blending in with normal traffic. Technical Report GIT-CC-04-13, College of Computing, Georgia Tech, 2004.

[13] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. *In 14th Usenix Security Symposium*, 2005.

[14] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. *In Recent Advances in Intrusion Detection*, 2005.

[15] C. Kruegel, T. Toth, and E. Kirda. Service specific anomaly detection for network intrusion detection. *In Proceedings of ACM SIGSAC*, 2002.

[16] C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. *In Proceedings of ACM CCS*, pages 251–261, 2003.

[17] Ktwo. Admmutate: Shellcode mutation engine. *http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz*, 2001.

[18] M. Mahoney. Network traffic anomaly detection based on packet bytes. *In Proceedings of ACM SIGSAC*, 2003.

[19] M. Mahoney and P.K. Chan. Learning nonstationary models of normal network traffic for detecting novel attacks. *In Proceedings of SIGKDD*, 2002.

[20] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. *In Proceeding of the IEEE Security and Privacy Conference*, 2005.

[21] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif. Misleading worm signature generators using deliberate noise injection. *In Proceedings of the IEEE Security and Privacy Conference*, 2006.

[22] T.H. Ptacek and T.N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. *Technical Report T2R-0Y6, Secure Networks, Inc.*, 1998.

[23] Rain Forest Puppy. A look at whisker's anti- ids tactics just how bad can we ruin a good thing? *www.wiretrip.net/rfp/txt/whiskerids.html*, 1999.

[24] S. Rubin, S. Jha, and B.P. Miller. Automatic generation and analysis of nids attacks. *In Annual Computer Security Applications Conference (ACSAC)*, 2004.

[25] M. Sedalo. Jempiscodes: Polymorphic shellcode generator. *www.shellcode.com.ar/en/proyectos.html*.

[26] D. Song. Fragroute: a tcp/ip fragmenter. *www.monkey.org/∼dugsong/fragroute*, 2002.

[27] Sal Stolfo. Personal communication. 2005.

[28] P. Szor. Advanced code evolution techniques and computer virus generator kits. *The Art of Computer Virus Research and Defense*, 2005.

[29] K.M.C. Tan, K.S. Killourhy, and R.A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. *In Recent Advances in Intrusion Detection*, 2002.

[30] T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract payload execution. *In Recent Advances in Intrusion Detection*, 2002.

[31] G. Vigna, W. Robertson, and D. Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. *In Proceedings of the ACM Conference on Computer and Communication Security (ACM CCS)*, pages 21–30, 2004.

[32] D. Wagner and D. Dean. Intrusion detection via static analysis. *In Proceeding of IEEE Symposium on Security and Privacy*, 2001.

[33] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. *In Proceedings of the ACM Conference on Computer and Communication Security (ACM CCS)*, 2002.

[34] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. *In the Proceedings of ACM SIGCOMM*, 2004.

[35] K. Wang and S. Stolfo. Anomalous payload-based network intrusion detection. *In Recent Advances in Intrusion Detection*, 2004.

[36] K. Wang and S. Stolfo. Anomalous payload-based worm detection and signature generation. *In Recent Advances in Intrusion Detection*, 2005.

[37] T. Yetiser. Polymorphic viruses: Implementation, detection, and protection. Technical report, VDS Advanced Research Group, 1993.

Notes

[1]Used by authors of PAYL to compare two models for convergence

6 APPENDIX

6.1 Proof of Optimal Padding for 1-gram Blending Attack

We prove that the padding calculated using Equation (7) is minimum for matching the 1-gram profile exactly.

Theorem 6.1 $\lambda_i \geq 0, \forall 1 \leq i \leq n$

Proof We prove the theorem by contradiction. Assume that for some j, $\lambda_j < 0$. Then from Equation (7), $\|\hat{w}\|(\delta f(x_j) - \hat{f}(x_j)) < 0$. Thus, $\delta < \frac{\hat{f}(x_j)}{f(x_j)}$. This contradicts Equation (5), therefore, all $\lambda_i \geq 0$. ▮

The frequency of a character x_i in the packet after padding is $\acute{f}(x_i) = \frac{\|\hat{w}\|\hat{f}(x_i)+\lambda_i}{\|\hat{w}\|}$. Using Equation (7) and Equation (4), $\acute{f}(x_i) = f(x_i)$. Thus, the final attack packet after padding has the exact target distribution, $f(x_i)$.

Theorem 6.2 *The padding calculated using Equation (7) is the minimum required padding to match frequencies exactly.*

Proof Suppose we perform padding using Equation (7). Suppose there exists another packet (say p, $\|p\| < \|\hat{w}\|$) with smaller padding and matches the frequencies exactly. Since $\lambda_k = 0$, the number of occurrences of x_k in p cannot decrease. Thus, frequency of x_k in packet p is $f_p(x_k) = \frac{\|\hat{w}\|\hat{f}(x_k)}{\|p\|} = \frac{\|\hat{w}\|f(x_k)}{\|p\|} > f(x_k)$. Thus, packet p does not match the normal frequencies exactly. Thus, we have reached a contradiction. ▮

6.2 Proof of Hardness of 2-gram Single-Byte Encoding

First, we look at the problem of evading a simple IDS that stores all the 2-grams present in the normal stream. While monitoring, it checks if all the 2-grams present in the traffic are also present in the normal 2-gram list. In the event that the IDS finds a 2-gram that was not present in normal traffic, IDS raises an alarm. Blending the attack packet with the normal traffic requires the attacker to transform the packet such that all the 2-grams in the packet after substitution are also present in the normal 2-gram list. Matching the frequencies of the tuples is at least as hard as the above simplified problem.

Suppose we have a normal traffic profile (N, T_N) and an attack packet description (M, T_M), where N and M is the set of normal and attack characters, respectively. T_N and T_M is the set of different 2-grams present in normal traffic and the attack, respectively. Also, the attacker is allowed to do only one-to-one substitution from M to N. Then, blending of the packet translates to finding a substitution S such that all the tuples in $S(w)$ are also present the normal profile. That is if $a_1a_2 \in T_M$, then $S(a_1a_2) \in T_N$.

Theorem 6.3 *The problem of finding a one-to-one substitution S to match 2-grams is NP-complete.*

Proof To prove that the problem is in NP-complete, we need to show that the problem is polynomial time verifiable and NP-hard.

Given a solution substitution S for the 2-gram matching problem, we can calculate $S(w)$ in $O(\|w\|)$ steps. For each 2-gram present in $S(w)$, checking if it is present in T_N can be done in $O(\|w\|.T_M)$ steps. Thus, this problem is poly-verifiable and consequently in NP.

To show that the problem is NP-hard, we reduce the problem of sub-graph isomorphism to substitution problem. A sub-graph isomorphism problem is that given two graphs $G(V, E)$ and $G'(V', E')$, decide whether G' is a sub-graph of G. Mathematically, we want to check if there is a mapping $S(V' \mapsto V)$, s.t. $\forall (v_1, v_2) \in E', (S(v_1), S(v_2)) \in E$.

Suppose, $N = V$. For each edge $e = (v_1, v_2) \in E$, add two 2-grams (v_1v_2, v_2v_1) in the normal profile (T_N). Suppose $M = V'$. For each edge $e' = (v_1, v_2) \in E'$, we add two 2-grams (v_1v_2, v_2v_1) in the attack profile (T_M).

If the above 2-gram matching problem has a solution, then we can find a mapping $S(V' \mapsto V)$ such that for all 2-grams $(a_1a_2) \in T_M$, $S(a_1a_2) \in T_N$. Since the 2-grams in T_M correspond to edges in G' and the 2-grams in T_N correspond to edges in G, the above statement suggests that $\forall e' \in G', S(e') \in G$. This means that graph G' is isomorphic to a sub-graph of G with mapping given by S.

Also, if there does not exist a solution to the 2-gram matching problem, then there does not exists a substitution S_t such that G' is a sub-graph of G after substitution. Otherwise, S_t will result in a successful 2-gram mapping.

Thus, the 2-gram matching problem is at least as hard as the sub-graph isomorphism problem. It is known that the sub-graph isomorphism problem is NP-complete. Also, we have already proved that the 2-gram matching problem is in NP. Thus, the 2-gram matching problem is NP-complete. ▮

Even if an IDS allows constant number of mismatches, it can be shown that the problem still remains NP-complete. This is followed by the result that sub-graph isomorphism with constant number of edge insertion, deletion, and substitution is also NP-complete. This means that an attacker cannot get the substitution that will match the normal profile with a small constant number of mismatched 2-grams. Also, the one-to-one substitution problem can be easily reduced to one-to-many substitution. Thus, solving one-to-many substitution is also hard.

Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection

Holger Dreger
TU München
dreger@in.tum.de

Anja Feldmann
TU München
feldmann@in.tum.de

Michael Mai
TU München
maim@in.tum.de

Vern Paxson
ICSI/LBNL
vern@icir.org

Robin Sommer
ICSI
robin@icir.org

Abstract

Many network intrusion detection systems (NIDS) rely on protocol-specific analyzers to extract the higher-level semantic context from a traffic stream. To select the correct kind of analysis, traditional systems exclusively depend on well-known port numbers. However, based on our experience, increasingly significant portions of today's traffic are not classifiable by such a scheme. Yet for a NIDS, this traffic is very interesting, as a primary reason for not using a standard port is to evade security and policy enforcement monitoring. In this paper, we discuss the design and implementation of a NIDS extension to perform dynamic application-layer protocol analysis. For each connection, the system first identifies potential protocols in use and then activates appropriate analyzers to verify the decision and extract higher-level semantics. We demonstrate the power of our enhancement with three examples: reliable detection of applications not using their standard ports, payload inspection of FTP data transfers, and detection of IRC-based botnet clients and servers. Prototypes of our system currently run at the border of three large-scale operational networks. Due to its success, the bot-detection is already integrated into a dynamic inline blocking of production traffic at one of the sites.

1 Introduction

Network intrusion detection systems (NIDSs) analyze streams of network packets in order to detect attacks and, more generally, violations of a site's security policy. NIDSs often rely on protocol-specific analyzers to extract the higher-level semantic context associated with a traffic stream, in order to form more reliable decisions about if and when to raise an alarm [38]. Such analysis can be quite sophisticated, such as pairing up a stream of replies with previously pipelined requests, or extracting the specific control parameters and data items associated with a transaction.

To select the correct analyzer for some traffic, a NIDS faces the challenge of determining which protocol is in use before it even has a chance to inspect the packet stream. To date, NIDSs have resolved this difficulty by assuming use of a set of well-known ports, such as those assigned by IANA [19], or those widely used by convention. If, however, a connection does not use one of these recognized ports—or misappropriates the port designated for a different application—then the NIDS faces a quandary: how does it determine the correct analyzer?

In practice, servers indeed do not always use the port nominally associated with their application, either due to benign or malicious intent. Benign examples include users who run Web or FTP servers on alternate ports because they lack administrator privileges. Less benign, but not necessarily malicious, examples include users that run servers offering non-Web applications on port 80/tcp in order to circumvent their firewall. In fact, some recently emerging application-layer protocols are *designed* to work without any fixed port, primarily to penetrate firewalls and escape administrative control. A prominent example is the voice-over-IP application Skype [2], which puts significant efforts into escaping restrictive firewalls. Sometimes such applications leverage a common protocol and its well-known port, like HTTP, to *tunnel* their payload not just through the firewall but even through application layer proxies. In these cases, analyzing the application's traffic requires first analyzing and stripping off the outer protocol before the NIDS can comprehend the semantics of the inner protocol. Similarly, we know from operational experience that attackers can attempt to evade security monitoring by concealing their traffic on non-standard ports or on ports assigned to different protocols: trojans installed on compromised hosts often communicate on non-standard ports; many botnets use the IRC protocol on ports other than 666x/tcp; and pirates build file-distribution networks using hidden FTP servers on ports other than 21/tcp.

It is therefore increasingly crucial to drive protocol-specific analysis using criteria other than ports. Indeed, a recent study [37] found that at a large university about 40% of the external traffic could not be classified by a port-based heuristic. For a NIDS, this huge amount of traffic is very interesting, as a primary reason for not using a standard port is to evade security and policy enforcement monitoring. Likewise, it is equally pressing to inspect whether traffic on standard ports indeed corresponds to the expected protocol. Thus, NIDSs need the capability of examining such traffic in-depth, including decapsulating an outer protocol layer in order to then examine the one tunneled inside it.

However, none of the NIDSs which are known to us, including Snort [34], Bro [31], Dragon [14], and IntruShield [20], use any criteria other than ports for their protocol-specific analysis. As an initial concession to the problem, some systems ship with signatures—characteristic byte-level payload patterns—meant to *detect* the use of a protocol on a none-standard port. But all only report the mere fact of finding such a connection, rather than adapting their analysis to the dynamically detected application protocol. For example, none of these systems can extract URLs from HTTP sessions on ports other than the statically configured set of ports.[1] With regards to decapsulating tunnels, a few newer systems can handle special cases, e.g., McAfee's IntruShield system [20] can unwrap the SSL-layer of HTTPS connections when provided with the server's private key. However, the decision that the payload *is* SSL is still based on the well-known port number of HTTPS.

In this paper we discuss the design, implementation, deployment, and evaluation of an extension to a NIDS to perform dynamic application-layer protocol analysis. For each connection, the system identifies the protocol in use and activates appropriate analyzers. We devise a general and flexible framework that (i) supports multiple ways to recognize protocols, (ii) can enable multiple protocol analyzers in parallel, (iii) copes with incorrect classifications by disabling protocol analyzers, (iv) can pipeline analyzers to dynamically decapsulate tunnels, and (v) provides performance sufficient for high-speed analysis.

We demonstrate the power our enhancement provides with three examples: (i) *reliable* detection of applications not using their standard ports, (ii) payload inspection of FTP data transfers, and (iii) detection of IRC-based botnet clients and servers. The prototype system currently runs at the border of the University of California, Berkeley (UCB), the Münchener Wissenschaftsnetz (Munich Scientific Network, MWN), and the Lawrence Berkeley National Laboratory (LBNL). These deployments have already exposed a significant number of security incidents, and, due to its success, the staff of MWN has integrated bot-detection into its operations, using it for dynamic inline blocking of production traffic.

The remainder of this paper is organized as follows: §2 presents the three network environments that we use for our study. In §3 we analyze the potential of non-port-based protocol detection and discuss the limitations of existing NIDSs. In §4 we present the design and implementation of our dynamic architecture and discuss the trade-offs one faces in practice. §5 demonstrates the benefits of the dynamic architecture with three example applications. In §6 we evaluate the performance of our implementation in terms of CPU usage and detection capabilities. Finally in §7 we summarize our experience.

2 Environments and Dataset

The impetus for performing protocol analysis free of any assumptions regarding applications using standard ports arose from our operational experiences with NIDSs at three large-scale network environments: the *University of California, Berkeley (UCB)*, the *Münchener Wissenschaftsnetz (Munich Scientific Network, MWN)* and the *Lawrence Berkeley National Laboratory (LBNL)* [10]. We found that increasingly significant portions of the traffic at these sites were not classifiable using well-known port numbers. Indeed, at UCB 40% of all packets fall into this category [37].

All three environments support high volumes of traffic. At UCB, external traffic currently totals about 5 TB/day, with three 2 Gbps uplinks serving about 45,000 hosts on the main campus plus several affiliated institutes. The MWN provides a 1 Gbps upstream capacity to roughly 50,000 hosts at two major universities along with additional institutes, totaling 1-3 TB a day. LBNL also utilizes a 1 Gbps upstream link, transferring about 1.5 TB a day for roughly 13,000 hosts.

Being research environments, the three networks' security policies emphasize relatively unfettered connectivity. The border routers impose only a small set of firewall restrictions (e.g., closing ports exploited by major worms). MWN uses a more restrictive set of rules in order to close ports used by the major peer-to-peer (P2P) applications; however, since newer P2P applications circumvent such port-based blocking schemes, MWN is moving towards a dynamic traffic filtering/shaping system. In a first step it leverages NAT gateways[16] used to provide Internet access to most student residences, and the IPPP2P system for detecting peer-to-peer traffic [21].

In §5 we report on our experiences with running three different example applications of our extended NIDS on live traffic. To enable a systematic evaluation (see §3.2 and §6), we captured a 24-hour *full* trace

[1]To keep our terminology simple, we will refer to a single fixed port when often this can be extended to a fixed set of ports.

at MWN's border router on October 11, 2005, using a high-performance Endace DAG capturing card [13]. The trace encompasses 3.2 TB of data in 6.3 billion packets and contains 137 million distinct connections. 76% of all packets are TCP. The DAG card did not report any packet losses.

3 Analysis of the Problem Space

Users have a variety of reasons for providing servicing on non-standard ports. For example, a site's policy might require private services (such as a Web server) to run on an unprivileged, often non-standard, port. Such private servers frequently do not run continuously but pop up from time to time, in contrast to business-critical servers. From our operational experience, in open environments such servers are common and not viewed as any particular problem. However, compromised computers often also run servers on non-standard ports, for example to transfer sometimes large volumes of pirated content. Thus, some servers on non-standard port are benign, others are malicious; the question of how to treat these, and how to distinguish among them, must in part be answered by the site's security policy.

In addition, users also use standard ports for running applications other than those expected on the ports, for example to circumvent security or policy enforcement measures such as firewalls, with the most prevalent example being the use of port 80/tcp to run P2P nodes. A NIDS should therefore not assume that every connection on HTTP's well-known port is indeed a communication using the HTTP protocol; or, even if it *is* well-formed HTTP, that it reflects any sort of "Web" access. The same problem, although often unintentional and not malicious, exists for protocols such as IRC. These are not assigned a well-known privileged port but commonly use a set of well-known unprivileged ports. Since these ports are unprivileged, other applications, e.g., an FTP data-transfer connection, may happen to pick one of these ports. A NIDS therefore may encounter traffic from a different application than the one the port number indicates. Accordingly the NIDS has to have a way to detect the application layer protocol actually present in order to perform application-specific protocol analysis.

3.1 Approaches to Application Detection

Besides using port numbers, two other basic approaches for identifying application protocols have been examined in the literature: (i) statistical analysis of the traffic within a connection, and (ii) locating protocol-specific byte patterns in the connection's payload.

Previous work has used an analysis of interpacket delays and packet size distribution to distinguish interactive applications like chat and remote-login from bulk-transfer applications such as file transfers [41]. In some particular contexts these techniques can yield good accuracy, for example to separate Web-chat from regular Web surfing [8]. In general, these techniques [29, 35, 23, 40], based on statistical analysis and/or or machine learning components, have proven useful for classifying traffic into broad classes such as interactive, bulk transfer, streaming, or transactional. Other approaches model characteristics of individual protocols by means of decision trees [39] or neural networks [12].

The second approach—using protocol-specific, byte-level payload patterns, or "signatures"—takes advantage of a popular misuse detection technique. Almost all virus-scanner and NIDSs incorporate signatures into their analysis of benign vs. malicious files or network streams. For protocol recognition, we can use such signatures to detect application-specific patterns, such as components of an HTTP request or an IRC login sequence. However, there is no guarantee that such a signature is comprehensive. If it fails to detect all instances of a given application, it exhibits *false negatives*. In addition, if it incorrectly attributes a connection to a given application, it exhibits *false positives*.

We can also combine these types of approaches, first using statistical methods (or manual inspection) to cluster connections, and then extracting signatures, perhaps via machine learning techniques [17]; or using statistical methods to identify some applications, and signatures to identify others [41] or to refine the classification, or to combine ports, content-signatures, and application-layer information [6].

In the context of NIDSs, signature-based approaches are particularly attractive because many NIDSs already provide an infrastructure for signature-matching (§3.3), and often signatures yield tighter protocol identification capabilities.

3.2 Potential of a Signature Set

To evaluate how often common protocols use non-standard ports, and whether signatures appear capable of detecting such uses, we examine a 24-hour full trace of MWN's border router, `mwn-full-packets`. To do so we use the large, open source collection of application signatures included with the *l7-filter* system [24]. To apply these signatures to our trace, we utilize the signature matching engine of the open source NIDS Bro [31, 38]. Rather than running the l7-filter system itself, which is part of the Linux netfilter framework [30], we convert the signatures into Bro's syntax, which gives us the advantages of drawing upon Bro's built-in trace processing, connection-oriented analysis, and powerful signature-matching engine. We note however that while Bro and l7-filter perform the matching in a similar way, varying internal semantics can lead to slightly different results, as with any two matching engines [38].

We begin by examining the breakdown of connections by the destination port seen in initial SYN packets. Table 1 shows all ports accounting for more than one percent of the connections. Note that for some ports the number of raw connections can be misleading due to the huge number of scanners and active worms, e.g., ports 445, 1042, and 1433. We consider a connection unsuccessful if it either does not complete an initial TCP handshake, or it does but does not transfer any payload. Clearly, we cannot identify the application used by such connections given no actual contents.

We make two observations. First, port-based protocol identification offers little assistance for most of the connections using unprivileged ports (totaling roughly 5.6 million connections). Second, the dominance of port 80 makes it highly attractive as a place for hiding connections using other applications. While an HTTP protocol analyzer might notice that such connections do not adhere to the HTTP protocol, we cannot expect that the analyzer will then go on to detect the protocol actually in use.

To judge if signatures can help improve application identification, for each of a number of popular apparent services (HTTP, IRC, FTP, and SMTP) we examined the proportion identified by the l7-filter signatures as indeed running that protocol. Table 2 shows that most of the successful connections trigger the expected signature match (thus, the signature quality is reasonable). Only for FTP we observe a higher percentage of false negatives. This can be improved using a better FTP signature. However, we also see that for each protocol we find matches for connections on unexpected ports, highlighting the need for closer inspection of their payload.

The differences in Table 2 do not necessarily all arise due to false negatives. Some may stem from connections without enough payload to accurately determine their protocol, or those that use a different protocol. Regarding this latter, Table 3 shows how often a different protocol appears on the standard ports of HTTP, IRC, FTP and SMTP.

While inspecting the results we noticed that a connection sometimes triggers more than one signature. More detailed analysis reveals that l7-filter contains some signatures that are too general. For example, the signature for the Finger protocol matches simply if the first two characters at the beginning of the connection are printable characters. Such a signature will be triggered by a huge number of connections not using Finger. Another example comes from the "whois" signature. Accordingly, the data in Table 3 ignores matches by these two signatures.

Port	HTTP	IRC	FTP	SMTP	Other	No sig.
80	92.2M	59	0	0	41.1K	1.2M
6665-6669	1.2K	71.7K	0	0	4.2	524
21	0	0	98.0K	2	2.3K	52.5K
25	459	2	749	1.4M	195	31.9K

Table 3: Signature-based detection vs. port-based detection for well-known ports (# connections).

Overall, the results show that the problem we pose does indeed already manifest operationally. Furthermore, because security analysis entails an adversary, what matters most is not the proportion of benign connections using ports other than those we might expect, but the prevalence of malicious connections doing so. We later discuss a number of such instances found operationally.

3.3 Existing NIDS Capabilities

Today's spectrum of intrusion detection and prevention systems offer powerful ways for detecting myriad forms of abuse. The simpler systems rely on searching for byte patterns within a packet stream, while the more complex perform extensive, stateful protocol analysis. In addition, some systems offer anomaly-based detection, comparing statistical characteristics of the monitored traffic against "normal network behavior," and/or specification-based detection, testing the characteristics against explicit specifications of allowed behavior.

For analyzing application-layer protocols, all systems of which we are aware depend upon port numbers.[2] While some can use signatures to *detect* other application-layer protocols, all only perform detailed protocol analysis for traffic identified via specific ports. Commercial systems rarely make details about their implementation available, and thus we must guess to what depth they analyze traffic. However, we have not seen an indication yet that any of them initiates stateful protocol analysis based on other properties than specific ports.

The most widely deployed open source NIDS, Snort [34], does not per se ship with signatures for detecting protocols. However the Snort user community constantly contributes new open signatures [4], including ones for detecting IRC and FTP connections. Traditionally, Snort signatures are raw byte patterns. Newer versions of Snort also support regular expressions. Another open source NIDS, Bro [31], ships with a *backdoor* [41] analyzer which follows two approaches. First, to detect interactive traffic it examines inter-packet intervals and packet size distributions. Second, for several

[2]`DSniff` [11] is a network sniffer that extracts protocol-specific usernames and passwords independent of ports. Its approach is similar to ours in that it uses a set of patterns to recognize protocols. It is however not a NIDS and does not provide any further payload analysis.

Port	Connections	% Conns.	Successful	% Success.	Payload [GB]	% Payload
80	97,106,281	70.82%	93,428,872	68.13	2,548.55	72.59
445	4,833,919	3.53%	8,398	0.01	0.01	0.00
443	3,206,369	2.34%	2,855,457	2.08	45.22	1.29
22	2,900,876	2.12%	2,395,394	1.75	59.91	1.71
25	2,533,617	1.85%	1,447,433	1.05	60.00	1.71
1042	2,281,780	1.66%	35	0.00	0.01	0.00
1433	1,451,734	1.06%	57	0.00	0.06	0.00
135	1,431,155	1.04%	62	0.00	0.00	0.00
< 1024	114,747,251	83.68%	101,097,769	73.73	2,775.15	79.05
≥ 1024	22,371,805	16.32%	5,604,377	4.08	735.62	20.95

Table 1: Ports accounting for more than 1% of the `mwn-full-packets` connections.

Method	HTTP	%	IRC	%	FTP	%	SMTP	%
Port (successful)	93,429K	68.14	75,876	0.06	151,700	0.11	1,447K	1.06
Signature	94,326K	68.79	73,962	0.05	125,296	0.09	1,416K	1.03
on expected port	92,228K	67.3	71,467	0.05	98,017	0.07	1,415K	1.03
on other port	2,126K	1.6	2,495	0.00	27,279	0.02	265	0.00

Table 2: Comparison of signature-based detection vs. port-based detection (# connections).

well-known protocols like HTTP, FTP and IRC, it scans the analyzed payload for hard-coded byte patterns. In addition, Bro features a signature matching engine [38] capable of matching the reassembled data stream of a connection against regular expression byte-patterns and leveraging the rich state of Bro's protocol decoders in the process. The engine allows for bidirectional signatures, where one byte pattern has to match on a stream in one direction and another in the opposite direction. The commercial IntruShield system by Network Associates is primarily signature-based and ships with signatures for application detection, including SSH and popular P2P protocols. The technical details and the signatures do not appear accessible to the user. Therefore, it is unclear which property of a packet/stream triggers which signature or protocol violation. We also have some experience with Enterasys' Dragon system. It ships with a few signatures to match protocols such as IRC, but these do not appear to then enable full protocol analysis.

3.4 NIDS Limitations

It is useful to distinguish between the capability of detecting that a given application protocol is in use, versus then being able to continue to analyze that instance of use. Merely detecting the use of a given protocol can already provide actionable information; it might constitute a policy violation at a site for which a NIDS could institute blocking without needing to further analyze the connection. However, such a coarse-grained "go/no-go" capability has several drawbacks:

1. In some environments, such a policy may prove too restrictive or impractical due to the sheer size and diversity of the site. As user populations grow, the likelihood of users wishing to run legitimate servers on alternate ports rises.

2. Neither approach to application detection (byte patterns or statistical tests) is completely accurate (see §3.2). Blocking false-positives hinders legitimate operations, while failing to block false-negatives hinders protection.

3. Protocols that use non-fixed ports (e.g., Gnutella) can only be denied or allowed. Some of these, however, have legitimate applications as well as applications in violation of policy. For example, BitTorrent [5] might be used for distributing open-source software. Or, while a site might allow the use of IRC, including on non-standard ports, it highly desires to analyze all such uses in order to detect botnets.

In addition, some protocols are fundamentally difficult to detect with signatures, for example unstructured protocols such as Telnet. For Telnet, virtually any byte pattern at the beginning is potentially legitimate. Telnet can only be detected heuristically, by looking for plausible login dialogs [31]. Another example is DNS, a binary protocol with no protocol identifier in the packet. The DNS header consists of 16-bit integers and bit fields which can take nearly arbitrary values. Thus, reliably detecting DNS requires checking the consistency across many fields. Similar problem exist for other binary protocols.

Another difficulty is that if an attacker knows the signatures, they may try to avoid the byte patterns that trigger the signature match. This means one needs "tight" signatures which comprehensively capture any use of a protocol for which an attacked end-system might engage. Finding such "tight" signatures can be particularly difficult due to the variety of end-system implementations and their idiosyncrasies.

4 Architecture

In this section we develop a framework for performing dynamic application-layer protocol analysis. Instead of a static determination of what analysis to perform based on port numbers, we introduce a processing path that dynamically adds and removes analysis components. The scheme uses a protocol detection mechanism as a trigger to activate analyzers (which are then given the entire traffic stream to date, including the portion already scanned by the detector), but these analyzers can subsequently decline to process the connection if they determine the trigger was in error. Currently, our implementation relies primarily on signatures for protocol detection, but our design allows for arbitrary other heuristics.

We present the design of the architecture in §4.1 and a realization of the architecture for the open-source NIDS Bro in §4.2. We finish with a discussion of the tradeoffs that arise in §4.3.

4.1 Design

Our design aims to achieve flexibility and power-of-expression, yet to remain sufficiently efficient for operational use. We pose the following requirements as necessary for these goals:

Detection scheme independence: The architecture must accommodate different approaches to protocol detection (§3.1). In addition, we should retain the possibility of using multiple techniques in parallel (e.g., complementing port-based detection with signature-based detection).

Dynamic analysis: We need the capability of dynamically enabling or disabling protocol-specific analysis at any time during the lifetime of a connection. This goal arises because some protocol detection schemes cannot make a decision upon just the first packet of a connection. Once they make a decision, we must trigger the appropriate protocol analysis. Also, if the protocol analysis detects a false positive, we must have the ability to stop the analysis.

Modularity: Reusable components allow for code reuse and ease extensions. This becomes particularly important for dealing with multiple network substacks (e.g., IP-within-IP tunnels) and performing in parallel multiple forms of protocol analysis (e.g., decoding in parallel with computing packet-size distributions).

Efficiency: The additional processing required by the extended NIDS capabilities must remain commensurate with maintaining performance levels necessary for processing high-volume traffic streams.

Customizability: The combination of analysis to perform needs to be easily adapted to the needs of the local security policy. In addition, the trade-offs within the analysis components require configuration according to the environment.

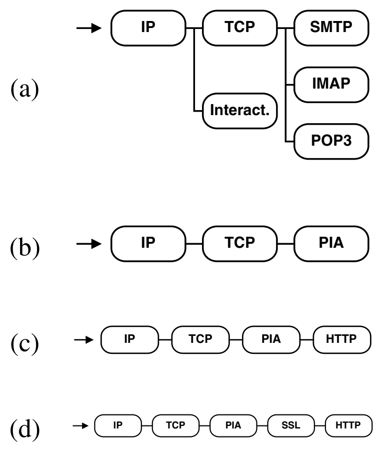


Figure 1: Example analyzer trees.

To address these requirements we switch from the traditional static data analysis path to a dynamic one inside the NIDS's core. Traditional port-based NIDSs decide at the time when they receive the first packet of each connection which analyses to perform. For example, given a TCP SYN packet with destination port 80, the NIDS will usually perform IP, TCP, and HTTP analysis for all subsequent packets of this flow. Our approach, on the other hand, relies on a per-connection data structure for representing the *data path*, which tracks what the system learns regarding what analysis to perform for the flow. If, for example, the payload of a packet on port 80/tcp—initially analyzed as HTTP—looks like an IRC session instead, we replace the HTTP analysis with IRC analysis.

We provide this flexibility by associating a tree structure with each connection. This tree represents the data path through various analysis components for all information transmitted on that connection (e.g., Figure 1(a)). Each node in this tree represents a self-contained unit of analysis, an *analyzer*. Each analyzer performs some kind of analysis on data received via an *input channel*, subsequently providing data via an *output channel*. The input channel of each node connects to an output channel of its data supplier (its predecessor in the data path tree). The input channel of the tree's root receives packets belonging to the connection/flow. Each intermediate node receives data via its input channel and computes analysis results, passing the possibly-transformed data to the next analyzer via its output channel.

Figure 1(a) shows an example of a possible analyzer tree for decoding email protocols. In this example, all analyzers (except `INTERACTIVE`) are responsible for the decoding of their respective network protocols. The packets of the connection first pass through the IP analyzer, then through the TCP analyzer. The output channel of the latter passes in replica to three analyzers for popular email protocols: SMTP, IMAP, and POP3. (Our architecture might instantiate such a tree for example if a signature match indicates that the payload looks like email but does not distinguish the application-layer protocol.) Note, though, that the analyzers need not correspond to a protocol, e.g., `INTERACTIVE` here, which examines inter-packet time intervals to detect surreptitious interactive traffic [41], performing its analysis in parallel to, and independent from, the TCP and email protocol analyzers.

To enable *dynamic* analysis, including analysis based on application-layer protocol identification, the analyzer tree changes over time. Initially, the analyzer tree of a new connection only contains those analyzers definitely needed. For example, if a flow's first packet uses TCP for transport, the tree will consist of an IP analyzer followed by a TCP analyzer.

We delegate application-layer protocol identification to a *protocol identification analyzer* (`PIA`), which works by applying a set of protocol detection heuristics to the data it receives. We insert this analyzer into the tree as a leaf-node after the TCP or UDP analyzer (see Figure 1(b)). Once the `PIA` detects a match for a protocol, it instantiates a child analyzer to which it then forwards the data it receives (see Figure 1(c)). However, the `PIA` also continues applying its heuristics, and if it finds another match it instantiates additional, or alternative, analyzers.

The analyzer tree can be dynamically adjusted throughout the entire lifetime of a connection by inserting or removing analyzers. Each analyzer has the ability to insert or remove other analyzers on its input and/or output channel. Accordingly, the tree changes over time. Initially the `PIA` inserts analyzers as it finds matching protocols. Subsequently one of the analyzers may decide that it requires support provided by a missing analyzer and instantiates it; for instance, an IRC analyzer that learns that the connection has a compressed payload can insert a decompression analyzer as its predecessor.

If an analyzer provides data via its output channel, selecting successors becomes more complicated, as not all analyzers (including the TCP analyzer) have the capability to determine the protocol to which their output data conforms. In this case the analyzer can choose to instantiate *another* `PIA` and delegate to it the task of further inspecting the data. Otherwise it can simply instantiate the appropriate analyzer; see Figure 1(d) for the example of a connection using HTTP over SSL.

Finally, if an analyzer determines that it cannot cope with the data it receives over its input channel (e.g., because the data does not conform to its protocol), it removes its subtree from the tree.

This analyzer-tree design poses a number of technical challenges, ranging from the semantics of "input channels", to specifics of protocol analyzers, to performance issues. We now address each in turn.

First, the semantics of "input channels" differ across the network stack layers: some analyzers examine packets (e.g., IP, TCP, and protocols using UDP for transport), while others require byte-streams (e.g., protocols using TCP for transport). As the `PIA` can be inserted into arbitrary locations in the tree, it must cope with both types. To do so, we provide two separate input channels for each analyzer, one for packet input and one for stream input. Each analyzer implements the channel(s) suitable for its semantics. For example, the TCP analyzer accepts packet input and reassembles it into a payload stream, which serves as input to subsequent stream-based analyzers. An RPC analyzer accepts both packet and stream input, since RPC traffic can arrive over both UDP packets and TCP byte streams.

Another problem is the difficulty—or impossibility—of starting a protocol analyzer in the middle of a connection. For example, an HTTP analyzer cannot determine the correct HTTP state for such a partial connection. However, most non-port-based protocol detection schemes can rarely identify the appropriate analyzer(s) upon inspecting just the first packet of a connection. Therefore it is important that the `PIA` buffers the beginning of each input stream, up to a configurable threshold (default 4KB in our implementation). If the `PIA` decides to insert a child analyzer, it first forwards the data in the buffer to it before forwarding new data. This gives the child analyzer a chance to receive the total payload if detection occurred within the time provided by the buffer. If instantiation occurs only after the buffer has overflowed, the `PIA` only instantiates analyzers capable of resynchronizing to the data stream, i.e., those with support for partial connections.

Finally, for efficiency the `PIA` requires very lightweight execution, as we instantiate at least one for every flow/connection. To avoid unnecessary resource consumption, our design factors out the user configuration, tree manipulation interface, and functions requiring permanent state (especially state independent of a connection's lifetime) into a single central management component which also instantiates the initial analyzer trees.

In summary, the approach of generalizing the processing path to an analyzer tree provides numerous new possibilities while addressing the requirements. We can: *(i)* readily plug in new protocol detection schemes via the `PIA`; *(ii)* dynamically enable and disable analyzers at any time (protocol semantics permitting); *(iii)* enable the user to customize and control the processing via an interface to the central manager; *(iv)* keep minimal the overhead of passing data along the tree branches; *(v)* support pure port-based analysis using a static analyzer tree installed at connection initialization; and *(vi)* support modularity by incorporating self-contained analyzers using a standardized API, which allows any protocol analyzer to also serve as a protocol *verifier*.

4.2 Implementation

We implemented our design within the open-source *Bro* NIDS, leveraging both its already existing set of protocol decoders and its signature-matching engine. Like other systems, Bro performs comprehensive protocol-level analysis using a static data path, relying on port numbers to identify application-layer protocols. However, its modular design encourages application-layer decoders to be mainly self-contained, making it feasible to introduce a dynamic analyzer structure as discussed in §4.1.

We implemented the `PIA`, the analyzer trees, and the central manager, terming this modification of Bro as `PIA-Bro`; for details see [26]. We use signatures as our primary protocol-detection heuristic (though see below), equipping the `PIA` with an interface to Bro's signature-matching engine such that analyzers can add signatures corresponding to their particular protocols. For efficiency, we restricted the signature matching to the data buffered by the `PIA`s; previous work[36, 28] indicates that for protocol detection it suffices to examine at most a few KB at the beginning of a connection. By skipping the tails, we can avoid performing pattern matching on the bulk of the total volume, exploiting the heavy-tailed nature of network traffic [32].

In addition to matching signatures, our implementation can incorporate other schemes for determining the right analyzers to activate. First, the `PIA` can still activate analyzers based on a user-configured list of well-known ports.[3] In addition, each protocol analyzer can register a specific detection function. The `PIA` then calls this function for any new data chunk, allowing the use of arbitrary heuristics to recognize the protocol. Finally, leveraging the fact that the central manager can store state, we also implemented a *prediction table* for storing anticipated future connections along with a corresponding analyzer. When the system eventually encounters one of these connections, it inserts the designated analyzer into the tree. (See §5.2 below for using this mechanism to inspect FTP data-transfer connections.) Together these mechanisms provide the necessary flexibility for the connections requiring dynamic detection, as well as good performance for the bulk of statically predictable connections.

As Bro is a large and internally quite complex system, we incrementally migrate its protocol analyzers to use the new framework. Our design supports this by allowing old-style and new-style data paths to coexist: for those applications we have adapted, we gain the full power of the new architecture, while the other applications remain usable in the traditional (static ports) way.

For our initial transition of the analyzers we have concentrated on protocols running on top of TCP. The Bro system already encapsulates its protocol decoders into separate units; we redesigned the API of these units to accommodate the dynamic analyzer structure. We have converted four of the system's existing application-layer protocol decoders to the new API: FTP, HTTP, IRC, and SMTP.[4] The focus on TCP causes the initial analyzer tree to always contain the IP and TCP analyzers. Therefore we can leverage the existing static code and did not yet have to adapt the IP and TCP logic to the new analyzer API. We have, however, already moved the TCP stream reassembly code into a separate "Stream" analyzer. When we integrate UDP into the framework, we will also adapt the IP and TCP components.

The Stream analyzer is one instance of a *support analyzer* which does not directly correspond to any specific protocol. Other support analyzers provide functionality such as splitting the payload-stream of text-based protocols into lines, or expanding compressed data.[5] We have not yet experimented with pipelining protocol analyzers such as those required for tunnel decapsulation, but intend to adapt Bro's SSL decoder next to enable us to analyze HTTPS and IMAPS in a pipelined fashion when we provide the system with the corresponding secret key.

[3]This differs from the traditional Bro, where the set of well-known ports is hard-coded.

[4]Note that it does not require much effort to convert an existing application-layer analyzer to the new API. For example, the SMTP analyzer took us about an hour to adapt.

[5]Internally, these support analyzers are implemented via a slightly different interface, see [26] for details.

4.3 Trade-Offs

Using the PIA architecture raises some important trade-offs to consider since protocol recognition/verification is now a multi-step process. First, the user must decide what kinds of signatures to apply to detect *potential* application-layer protocols. Second, if a signature matches it activates the appropriate protocol-specific analyzer, at which point the system must cope with possible false positives; when and how does the analyzer fail in this case? Finally, we must consider how an attacker can exploit these trade-offs to subvert the analysis.

The first trade-off involves choosing appropriate signatures for the protocol detection. On the one hand, the multi-step approach allows us to *loosen* the signatures that initially detect protocol candidates. Signatures are typically prone to false alerts, and thus when used to generate alerts need to be specified as tight as possible—which in turn very often leads to false negatives, i.e., undetected protocols in this context. However, by relying on analyzers *verifying* protocol conformance after a signature match, false positives become more affordable. On the other hand, signatures should not be *too* lose: having an analyzer inspect a connection is more expensive than performing pure pattern matching. In addition, we want to avoid enabling an attacker to trigger expensive protocol processing by deliberately crafting bogus connection payloads.

Towards these ends, our implementation uses bidirectional signatures [38], which only match if *both* endpoints of a connection appear to participate in the protocol. If an attacker only controls one side (if they control both, we are sunk in many different ways), they thus cannot force activation of protocol analyzers by themselves. In practice, we in fact go a step further: before assuming that a connection uses a certain protocol, the corresponding analyzer must also *parse* something meaningful for both directions. This significantly reduces the impact of false positives. Figure 2 shows an example of the signature we currently use for activating the HTTP analyzer. (We note that the point here is not about signature quality; for our system, signatures are just one part of the NIDS's configuration to be adapted to the user's requirements.)

Another trade-off to address is *when* to decide that a connection uses a certain protocol. This is important if the use of a certain application violates a site's security policy and should cause the NIDS to raise an alert. A signature-match triggers the activation of an analyzer that analyzes and verifies the protocol usage. Therefore, before alerting, the system waits until it sees that the analyzer is capable of handling the connection's payload. In principle, it can only confirm this with certainty once the connection completes. In practice, doing so will delay alerts significantly for long-term connections. Therefore our implementation assumes that if the analyzer can parse the connection's *beginning*, the rest of the payload will also adhere to the same protocol. That is, our system reports use of a protocol if the corresponding analyzer is (still) active after the exchange of a given volume of payload, or a given amount of time passes (both thresholds are configurable).

Another trade-off stems from the question of protocol verification: at what point should an analyzer indicate that it *cannot* cope with the payload of a given connection? Two extreme answers: (i) reject immediately when something occurs not in accordance with the protocol's definition, or (ii) continue parsing come whatever may, in the hope that eventually the analyzer can resynchronize with the data stream. Neither extreme works well: real-world network traffic often stretches the bounds of a protocol's specification, but trying to parse the entire stream contradicts the goal of verifying the protocol. The right balance between these extremes needs to be decided on a per-protocol basis. So far, we have chosen to reject connections if they violate basic protocol properties. For example, the FTP analyzer complains if it does not find a numeric reply-code in the server's response. However, we anticipate needing to refine this decision process for instances where the distinction between clear noncompliance versus perhaps-just-weird behavior is less crisp.

Finally, an attacker might exploit the specifics of a particular analyzer, avoiding detection by crafting traffic in a manner that the analyzer believes reflects a protocol violation, while the connection's other endpoint still accepts or benignly ignores the data. This problem appears fundamental to protocol detection, and indeed is an instance of the more general problem of evasion-by-ambiguity [33, 18], and, for signatures, the vulnerability of NIDS signatures to attack variants. To mitigate this problem, we inserted indirection into the decision process: in our implementation, an analyzer *never* disables itself, even if it fails to parse its inputs. Instead, upon severe protocol violations it generates Bro events that a user-level policy script then acts upon. The default script is fully customizable, capable of extension to implementing arbitrary complex policies such as disabling the analyzer only after repeated violations. This approach fits with the Kerkhoff-like principle used by the Bro system: the code is open, yet sites code their specific policies in user-level scripts which they strive to keep secret.

5 Applications

We now illustrate the increased detection capabilities that stem from realizing the PIA architecture within Bro, using three powerful example applications: (i) reliable detection of applications running on non-standard ports,

```
signature http_server {                     # Server-side signature
  ip-proto == tcp                           # Examine TCP packets.
  payload /^HTTP\/[0-9]/                    # Look for server response.
  tcp-state responder                       # Match responder-side of connection.
  requires-reverse-signature http_client # Require client-side signature as well.
  enable "http"                             # Enable analyzer upon match.
}
signature http_client {                     # Client-side signature
  ip-proto == tcp                           # Examine TCP packets.
  payload /^[[:space:]]*GET[[:space:]]*/ # Look for requests [simplified]
  tcp-state originator                      # Match originator-side of connection.
}
```

Figure 2: Bidirectional signature for HTTP.

(ii) payload inspection of FTP data transfers, and (iii) detection of IRC-based bot clients and servers. All three schemes run in day-to-day operations at UCB, MWN, and LBNL (see §2), where they have already identified a large number of compromised hosts which the sites' traditional security monitoring could not directly detect.

5.1 Detecting Uses of Non-standard Ports

As pointed out earlier, a `PIA` architecture gives us the powerful ability to verify protocol usage and extract higher-level semantics. To take advantage of this capability, we extended the reporting of `PIA-Bro`'s analyzers. Once the NIDS knows which protocol a connection uses, it can leverage this to extract more semantic context. For example, HTTP is used by a wide range of other protocols as a transport protocol. Therefore, an alert such as "connection uses HTTP on a non-standard port 21012", while useful, does not tell the whole story; we would like to know *what* that connection then does. We extended `PIA-Bro`'s HTTP analysis to distinguish the various protocols using HTTP for transport by analyzing the HTTP dialog. Kazaa, for example, includes custom headers lines that start with `X-Kazaa`. Thus, when this string is present, the NIDS generates a message such as "connection uses Kazaa on port 21021". We added patterns for detecting Kazaa, Gnutella, BitTorrent, Squid, and SOAP applications running over HTTP. In addition, the HTTP analyzer extracts the "Server" header from the HTTP responses, giving an additional indication of the underlying application.

We currently run the dynamic protocol detection for FTP, HTTP, IRC, and SMTP on the border routers of all three environments, though here we primarily report on experiences at UCB and MWN. As we have particular interest in the use of non-standard ports, and to reduce the load on `PIA-Bro`, we exclude traffic on the analyzers' well-known ports[6] from our analysis. (This setup prevents `PIA-Bro` from finding some forms of port abuse, e.g., an IRC connection running on the HTTP port. We postpone this issue to §6.)

At both UCB and MWN, our system quickly identified many servers[7] which had gone unnoticed. At UCB, it found within a day 6 internal and 17 remote FTP servers, 568/54830 HTTP servers (!), 2/33 IRC servers, and 8/8 SMTP servers running on non-standard ports. At MWN, during a similar period, we found 3/40 FTP, 108/18844 HTTP, 3/58 IRC, and 3/5 SMTP servers.

For FTP, IRC, and SMTP we manually checked whether the internal hosts were indeed running the detected protocol; for HTTP, we verified a subset. Among the checked servers we found only one false positive: `PIA-Bro` incorrectly flagged one SMTP server due to our choice regarding how to cope with false positives: as discussed in §4.3, we choose to not wait until the end of the connection before alerting. In this case, the SMTP analyzer correctly reported a protocol violation for the connection, but it did so only *after* our chosen maximal interval of 30 seconds had already passed; the server's response took quite some time. In terms of connections, HTTP is, not surprisingly, the most prevalent of the four protocols: at UCB during the one-day period, 99% of the roughly 970,000 reported off-port connections are HTTP. Of these, 28% are attributed to Gnutella, 22% to Apache, and 12% to Freechal [15]. At MWN, 92% of the 250,000 reported connections are HTTP, and 7% FTP (of these 70% were initiated by the same host). Of the HTTP connections, roughly 21% are attributed to BitTorrent, 20% to Gnutella, and 14% to SOAP.

That protocol analyzers can now extract protocol semantics not just for HTTP but also for the other protocols proves to be quite valuable. `PIA-Bro` generates detailed protocol-level log files for all connections. A short glance at, for example, an FTP log file quickly reveals whether an FTP server deserves closer attention. Figure 3 shows an excerpt of such a log file for an ob-

[6]For "well-known" we consider those for which a traditional Bro triggers application-layer analysis. These are port 21 for FTP, ports 6667/6668 for IRC, 80/81/631/1080/3128/8000/8080/8888 for HTTP (631 is IPP), and 25 for SMTP. We furthermore added 6665/6666/6668/6669/7000 for IRC, and 587 for SMTP as we encountered them on a regular basis. To further reduce the load on the monitor machines, we excluded a few high volume hosts, including the PlanetLab servers at UCB and the heavily accessed `leo.org` domain at MWN.

[7]In this context, a *server* is an IP address that accepts connections and participates in the protocol exchange. Due to NAT address space, we may underestimate or overestimate the number of actual hosts.

```
xxx.xxx.xxx.xxx/2373 > xxx.xxx.xxx.xxx/5560 start
response (220 Rooted Moron Version 1.00 4 WinSock ready...)
USER ops (logged in)
SYST (215 UNIX Type: L8)
[...]
LIST -al (complete)
TYPE I (ok)
SIZE stargate.atl.s02e18.hdtv.xvid-tvd.avi (unavail)
PORT xxx,xxx,xxx,xxx,xxx,xxx (ok)
*STOR stargate.atl.s02e18.hdtv.xvid-tvd.avi, NOOP (ok)
ftp-data video/x-msvideo 'RIFF (little-endian) data, AVI'
[...]
response (226 Transfer complete.)
[...]
QUIT (closed)
```

Figure 3: Application-layer log of an FTP-session to a compromised server (anonymized/edited for clarity).

viously compromised host at MWN. During a two-week period, we found such hosts in both environments, although UCB as well as MWN already deploy Snort signatures supposed to detect such FTP servers.

With PIA-Bro, any protocol-level analysis automatically extends to non-standard ports. For example, we devised a detector for HTTP proxies which matches HTTP requests going into a host with those issued by the same system to external systems. With the traditional setup, it can only report proxies on well-known ports; with PIA-Bro in place, it has correctly identified proxies inside the UCB and MWN networks running on different ports;[8] two of them were world-open.

It depends on a site's policy whether offering a service on a non-standard port constitutes a problem. Both university environments favor open policies, generally tolerating offering non-standard services. For the internal servers we identified, we verified that they meet at least basic security requirements. For all SMTP servers, for example, we ensured that they do not allow arbitrary relaying. One at MWN which did was quickly closed after we reported it, as were the open HTTP proxies.

5.2 Payload Inspection of FTP Data

According to the experience of network operators, attackers often install FTP servers on non-standard ports on machines that they have compromised. PIA-Bro now not only gives us a reliable way to detect such servers but, in addition, can *examine* the transferred files. This is an impossible task for traditional NIDSs, as FTP is a protocol for which for the data-transfer connections by design use arbitrary port combinations. For security monitoring, inspecting the *transferred* data for files exchanged via non-standard-port services enables alerts on sensitive files such as system database accesses or download/upload of virus-infected files. We introduced a new *file analyzer* to perform such analysis for FTP data connections, as well as for other protocols used to transfer files. When PIA-Bro learns, e.g., via its analysis of the control session, of an upcoming data transfer, it adds the expected connection to the dynamic prediction table (see §4.2). Once this connection is seen, the system instantiates a file analyzer, which examines the connection's payload.

[8]As observing both internal as well as outgoing requests at *border* is rather unusual, this detection methodology generally detects proxies other than the site's intended ones.

The file analyzer receives the file's full content as a reassembled stream and can utilize any file-based intrusion detection scheme. To demonstrate this capability, our file-type identification for PIA-Bro leverages libmagic [25], which ships with a large library of file-type characteristics. This allows PIA-Bro to log the file-type's textual description as well as its MIME-type as determined by libmagic based on the payload at the beginning of the connection. Our extended FTP analyzer logs—and potentially alerts on—the file's content type. Figure 3 shows the result of the file type identification in the ftp-data line. The NIDS categorizes the data transfer as being of MIME type video/x-msvideo and, more specifically, as an AVI movie. As there usually are only a relatively small number of ftp-data connections, this mechanism imposes quite minimal performance overhead.

We envision several extensions to the file analyzer. One straight-forward improvement, suggested to us by the operators at LBNL, is to match a file's name with its actual content (e.g., a file picture.gif requested from a FTP server can turn out to be an executable). Another easy extension is the addition of an interface to a virus checker (e.g., Clam AntiVirus [7]). We also plan to adapt other protocol analyzers to take advantage of the file analyzer, such as TFTP (once PIA-Bro has support for UDP) and SMTP. TFTP has been used in the past by worms to download malicious code [3]. Similarly, SMTP can pass attachments to the file analyzer for inspection. SMTP differs from FTP in that it transfers files in-band, i.e., inside the SMTP session, rather than out-of-band over a separate data connection. Therefore, for SMTP there is no need to use the dynamic prediction table. Yet, we need the capabilities of PIA-Bro to pipeline the analyzers: first the SMTP analyzer strips the attachments' MIME-encoding, then the file analyzer inspects the file's content.

5.3 Detecting IRC-based Botnets

Attackers systematically install trojans together with *bots* for remote command execution on vulnerable systems. Together, these form large *botnets* controlled by a human *master* that communicates with the bots by sending commands. Such commands can be to flood a victim, send spam, or sniff confidential information such as passwords. Often, thousands of individual bots are controlled by a single master [1], constituting one of the largest security threats in today's Internet.

The IRC protocol [22] is a popular means for communication within botnets as it has some appealing properties for remote control: it provides *public channels* for one-to-many communication, with *channel topics* well-suited for holding commands; and it provides *private channels* for one-to-one communication.

It is difficult for a traditional NIDS to reliably detect members of IRC-based botnets. Often, the bots never connect to a standard IRC server—if they did they would be easy to track down—but to a separate *bot-server* on some non-IRC port somewhere in the Internet. However, users also sometimes connect to IRC servers running on non-standard ports for legitimate (non-policy-violating) purposes. Even if a traditional NIDS has the capability of detecting IRC servers on non-standard ports, it lacks the ability to then distinguish between these two cases.

We used `PIA-Bro` to implement a reliable bot-detector that has already identified a significant number of bot-clients at MWN and UCB. The detector operates on top of the IRC analyzer and can thus perform protocol-aware analysis of *all* detected IRC sessions. To identify a bot connection, it uses three heuristics. First, it checks if the client's nickname matches a (customizable) set of regular expression patterns we have found to be used by some botnets (e.g., a typical botnet "nick" identifier is `[0]CHN|3436036`). Second, it examines the channel topics to see if it includes a typical botnet command (such as `.advscan`, which is used by variants of the SdBot family[1]). Third, it flags new clients that establish an IRC connection to an already identified bot-server as bots. The last heuristic is very powerful, as it leverages the state that the detector accumulates over time and does not depend on any particular payload pattern. Figure 4 shows an excerpt of the list of known bots and bot-servers that one of our operational detectors maintains. This includes the server(s) contacted as well as the timestamp of the last alarming IRC command. (Such timestamps aid in identifying the owner of the system in NAT'd or DHCP environments.) For the servers, the list contains channel information, including topics and passwords, as well as the clients that have contacted them.

At MWN the bot-detector quickly flagged a large number of bots. So far, it has identified more than 100 distinct local addresses. To exclude the danger of false positives, we manually verified a subset. To date, we have not encountered any problems with our detection. Interestingly, at UCB there are either other kinds of bots, or not as many compromised machines; during a two-week time period we reported only 15 internal hosts to the network administrators. We note that the NIDS, due to only looking for patterns of *known* bots, certainly misses victims; this is the typical drawback of such a misuse-detection approach, but one we can improve over time as we learn more signatures through other means.

```
Detected bot-servers:
IP1 - ports 9009,6556,5552 password(s) <none> last 18:01:56
 channel #vec:
 topic ".asc pnp 30 5 999 -b -s|.wksescan 10 5 999 -b -s|[...]"
 channel #hv:
 topic ".update http://XXX/image1.pif f'', password(s) XXX"
[...]
Detected bots:
IP2 - server IP3 usr 2K-8006 nick [P00|DEU|59228] last 14:21:59
IP4 - server IP5 usr XP-3883 nick [P00|DEU|88820] last 19:28:12
[...]
```

Figure 4: Excerpt of the set of detected IRC bots and bot-servers (anonymized/edited for clarity).

Of the detected bots at MWN, only five used static IP addresses, while the rest used IP addresses from a NAT'd address range, indicating that most of them are private, student-owned machines. It is very time-consuming for the MWN operators to track down NAT'd IP addresses to their actual source. Worse, the experience at MWN is that even if they do, many of the owners turn out to not have the skills to remove the bot. Yet, it is important that such boxes cannot access the Internet.

The MWN operators accomplish this with the help of our system. They installed a blocking system for MWN's NAT subnets to which we interface with our system. The operators have found the system's soundness sufficiently reliable for flagging bots that they enabled it to block all reported bots *automatically*. They run this setup operationally, and so far without reporting to us any complaints. In the beginning, just after our system went online, the average number of blocked hosts increased by 10-20 addresses. After about two weeks of operation, the number of blocked hosts has almost settled back to the previous level, indicating that the system is effective: the number of bots has been significantly reduced.

Finally, we note that our detection scheme relies on the fact that a bot uses the IRC protocol in a fashion which conforms to the standard IRC specification. If the bot uses a custom protocol dialect, the IRC analyzer might not be able to parse the payload. This is a fundamental problem similar to the one we face if a bot uses a proprietary protocol. More generally we observe that the setting of seeing malicious clients *and* servers violates an important assumption of many network intrusion detection systems: an attacker does not control both endpoints of a connection [31]. If he does, any analysis is at best a heuristic.

6 Evaluation

We finish with an assessment of the performance impact of the `PIA` architecture and a look at the efficacy of the multi-step protocol recognition/verification process. The evaluation confirms that our implementation of the PIA framework does not impose an undue performance overhead.

6.1 CPU Performance

To understand the performance impact of the PIA extensions to Bro, we conduct CPU measurements for both the unmodified Bro (developer version 1.1.52), referred to as Stock-Bro, and PIA-Bro running on the first 66 GB of the mwn-full-packets trace which corresponds to 29 minutes of traffic. (This trace again excludes the domain leo.org.) The trace environment consists of a dual-Opteron system with 2GB RAM, running FreeBSD 6.0.

In addition to the processing of the (default) Stock-Bro configuration, PIA-Bro must also perform four types of additional work: (i) examining *all* packets; (ii) performing *signature matches* for many packets; and (iii) buffering and reassembling the beginnings of all streams to enable reliable protocol detection; (iv) performing application-layer protocol analysis on additionally identified connections. In total, these constitute the cost we must pay for PIA-Bro's additional detection capabilities.

To measure the cost of each additional analysis element, we enable them one by one, as reported in Table 4. We begin with basic analysis (Config-A): Bro's generation of one-line connection summaries, as well as application-layer protocol analysis for FTP, HTTP, IRC, SMTP connections, as identified via port numbers. The first line reports CPU times for both versions of Bro performing this regular analysis, and the second line when we also enable Bro signatures corresponding to those used by PIA-Bro for protocol detection. We find the runtimes of the two systems quite similar, indicating that our implementation of the PIA architecture does not add overhead to Bro's existing processing. (The runtime of PIA-Bro is slightly less than Stock-Bro due to minor differences in their TCP bytestream reassembly process; this also leads PIA-Bro to make slightly fewer calls to Bro's signature engine for the results reported below. The runtimes of both systems exceed the duration of the trace, indicating that we use a configuration which, in this environment, requires multiple NIDS instances in live operation.)

With Config-A, the systems only need to process a subset of all packets: those using the well-known ports of the four protocols, plus any with TCP SYN/FIN/RST control packets (which Bro uses to generate generic TCP connection summaries). Bro uses a BPF [27] filter to discard any other packets. However, PIA-Bro cannot use this filtering because by its nature it needs to examine *all* packets. This imposes a significant performance penalty, which we assess in two different ways.

First, we prefilter the trace to a version containing only those packets matched by Bro's BPF filter, which in this case results in a trace just under 60% the size of the original. Running on this trace rather than the original approximates the benefit Bro obtains when executing on systems that use in-kernel BPF filtering for which captured packets must be copied to user space but discarded packets do not require a copy. The Table shows these timings as Config-A'. We see that, for this environment and configuration, this cost for using PIA is minor, about 3.5%.

Second, we manually change the filters of both systems to include all TCP packets (Config-B). The user time increases by a fairly modest 7.5% for Stock-Bro and 7.4% for PIA-Bro compared to Config-B. Note that here we are not yet enabling PIA-Bro's additional functionality, but are only assessing the cost to Bro of processing the entire packet stream using the above configuration; this entails little extra work for Bro since it does not perform application analysis on the additional packets.

PIA-Bro's use of signature matching also imposes overhead. While most major NIDSs rely on signature matching, the Bro system's default configuration does not. Accordingly, applying the PIA signatures to the packet stream adds to the system's load. To measure its cost, we added signature matching to the systems' configuration (second line of the table, as discussed above). The increase compared to Config-A is about 15–16%.

When we activate signatures for Config-B, we obtain Config-C, which now enables essentially the full PIA functionality. This increases runtime by 24–27% for Stock-Bro and PIA-Bro, respectively. Note that by the comparison with Stock-Bro running equivalent signatures, we see that capturing the entire packet stream and running signatures against it account for virtually all of the additional overhead PIA-Bro incurs.

As the quality of the signature matches improves when PIA-Bro has access to the reassembled payload of the connections, we further consider a configuration based on Config-C that also reassembles the data which the central manager buffers. This configuration only applies to PIA-Bro, for which it imposes a performance penalty of 1.2%. The penalty is so small because most packets arrive in order [9], and we only reassemble the first 4KB (the PIA buffer size).

As we can detect most protocols within the first KBs (see §4.2), we also evaluated a version of PIA-Bro that restricts signature matching to only the first 4KB. This optimization, which we annotate as PIA-Bro-M4K, yields a performance gain of 16.2%. Finally, adding reassembly has again only a small penalty (2.1%).

		Stock-Bro	PIA-Bro	PIA-Bro-M4K
Config-A	Standard	3335	3254	—
	Standard + sigs	3843	3778	—
Config-A′	Standard	3213	3142	—
Config-B	All TCP pkts	3584	3496	—
Config-C	All TCP pkts + sigs	4446	4436	3716
	All TCP pkts + sigs + reass.	—	4488	3795

Table 4: CPU user times on subset of the trace (secs; averaged over 3 runs each; standard deviation always < 13s).

In summary, for this configuration we can obtain nearly the full power of the PIA architecture (examining all packets, reassembling and matching on the first 4KB) at a performance cost of about 13.8% compared to Stock-Bro. While this is noticeable, we argue that the additional detection power provided merits the expenditure. We also note that the largest performance impact stems from applying signature matching to a large number of packets, for which we could envision leveraging specialized hardware to speed up. Finally, because we perform dynamic protocol analysis on a per-connection basis, the approach lends itself well to front-end load-balancing.

6.2 Detection Performance

We finish with a look at the efficacy of the PIA architecture's multi-step analysis process. To do so, we ran PIA-Bro with all adapted analyzers (HTTP, FTP, IRC, SMTP) on the 24-hour mwn-full-packets trace, relying only our bidirectional PIA-signatures for protocol detection, i.e., no port based identification. (Note that as these signatures differ from the L7-signatures used in §3, the results are not directly comparable.) PIA-Bro verifies the detection as discussed in §4.3, i.e., when the connection has either run for 30 seconds or transferred 4 KB of data (or terminated).

Our goal is to understand the quality of its detection in terms of false positives and false negatives. In trading off these two, we particularly wish to minimize false positives, as our experience related in §5 indicates that network operators strongly desire actionable information when reporting suspected bot hosts or surreptitious servers.

Table 5 breaks down PIA-Bro's detections as follows. The first column shows how often (i) a protocol detection signature flagged the given protocol as running on a non-standard port, for which (ii) the corresponding analyzer verified the detection. With strong likelihood, these detections reflect actionable information.

The second and third columns list how often the analyzer did *not* agree with the detection, but instead rejected the connection as exhibiting the given protocol, for

	Detected and verified non-std. port	Rejected by analyzer non std. port	Rejected by analyzer std. port
HTTP	1,283,132	21,153	146,202
FTP	14,488	180	1,792
IRC	1,421	91	3
SMTP	69	0	1,368

Table 5: # of connections with detection and verification.

non-standard and standard ports, respectively. Thus, the second column highlights the role the analyzer plays in reducing false positives; had we simply employed signatures without subsequent verification, then in these cases we would have derived erroneous information.

The third column, on the other hand, raises questions regarding to what degree our protocol detection might be missing instances of given protocols. While we have not yet systematically assessed these rejections, those we have manually inspected have generally revealed either a significant protocol failure, or indeed an application other than that associated with the standard port. Examples of the former include errors in HTTP headers, non-numeric status codes in FTP responses, mismatches in SMTP dialogs between requests and responses, use of SMTP reply codes beyond the assigned range, and extremely short or mismatched IRC replies.

While we detect a large number of verified connections on non-standard ports—with the huge number of HTTP connections primarily due to various P2P applications—for this trace the only instance we found of a different protocol running on a privileged standard port was a (benign) IRC connection running on 80/tcp. On the unprivileged ports used for IRC, however, we found a private Apache HTTP server, a number of video-on-demand servers, and three FTP servers used for (likely illicit) music-sharing. (Note that, different than in §3.2, when looking for protocols running on standard ports, we can only detect instances of FTP, HTTP, IRC, and SMTP; also, protocols running *on top* of HTTP on port 80 are not reported.)

Finally, Figure 5 shows the diversity of the non-standard ports used by different types of servers. The

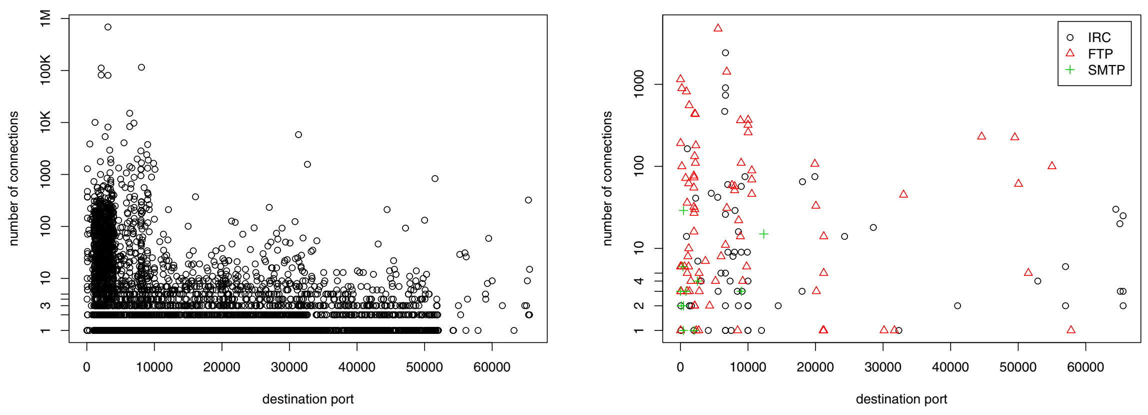


Figure 5: Connections using the HTTP (left) and the IRC, FTP, SMTP (right) protocol.

x-axis gives the port number used and the y-axis the number of connections whose servers resided on that port (log-scaled). The 22,647 HTTP servers we detected used 4,024 different non-standard ports, some involving more than 100,000 connections. We checked the top ten HTTP ports (which account for 88% of the connections) and found that most are due to a number of PlanetLab hosts (ports 312X, 212X), but also quite a large number are due to P2P applications, with Gnutella (port 6346) contributing the largest number of distinct servers. Similar observations, but in smaller numbers, hold for IRC, FTP, and SMTP, for which we observed 60, 81, and 11 different non-standard server ports, respectively. These variations, together with the security violations we discussed in §5, highlight the need for dynamic protocol detection.

7 Conclusion

In this paper we have developed a general NIDS framework which overcomes the limitations of traditional, port-based protocol analysis. The need for this capability arises because in today's networks an increasing share of the traffic resists correct classification using TCP/UDP port numbers. For a NIDS, such traffic is particularly interesting, as a common reason to avoid well-known ports is to evade security monitoring and policy enforcement. Still, today's NIDSs rely exclusively on ports to decide which higher-level protocol analysis to perform.

Our framework introduces a dynamic processing path that adds and removes analysis components as required. The scheme uses protocol detection mechanisms as triggers to activate analyzers, which can subsequently decline to process the connection if they determine the trigger was in error. The design of the framework is independent of any particular detection scheme and allows for the addition/removal of analyzers at arbitrary times. The design provides a high degree of modularity, which allows analyzers to work in parallel (e.g., to perform independent analyses of the same data), or in series (e.g., to decapsulate tunnels).

We implemented our design within the open-source Bro NIDS. We adapted several of the system's key components to leverage the new framework, including the protocol analyzers for HTTP, IRC, FTP, and SMTP, as well as leveraging Bro's signature engine as an efficient means for performing the initial protocol detection that is then verified by Bro's analyzers.

Prototypes of our extended Bro system currently run at the borders of three large-scale operational networks. Our example applications—reliable recognition of uses of non-standard ports, payload inspection of FTP data transfers, and detection of IRC-based botnet clients and servers—have already exposed a significant number of security incidents at these sites. Due to its success, the MWN site has integrated our bot-detection into dynamic blocking of production traffic.

In the near future, we will migrate the remainder of Bro's analyzers to the new framework. From our experiences to date, it appears clear that using dynamic protocol analysis operationally will significantly increase the number of security breaches we can detect.

8 Acknowledgments

We would like to thank the Lawrence Berkeley National Laboratory; the Leibniz-Rechenzentrum, München; and the University of California, Berkeley. We would also like to thank Deti Fliegl, John Ives, Jason Lee, and Brian Tierney. This work was supported by a fellowship within the Postdoc-Programme of the German Academic Exchange Service (DAAD), by the US National Science Foundation under grants ITR/ANI-0205519 and NSF-0433702, and by a grant from the Bavaria California Technology Center, for which we are grateful.

References

[1] T. H. P. . R. Alliance. Know your enemy: Tracking botnets. `http://www.honeynet.org/papers/bots`, 2005.

[2] S. A. Baset and H. Schulzrinne. An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol. In *Proc. IEEE Infocom 2006*, 2006.

[3] CERT Advisory CA-2003-20 W32/Blaster worm.

[4] BleedingSnort. `http://bleedingsnort.com`.

[5] BitTorrent. `http://www.bittorrent.com`.

[6] T. Choi, C. Kim, S. Yoon, J. Park, B. Lee, H. Kim, H. Chung, and T. Jeong. Content-aware Internet Application Traffic Measurement and Analysis. In *Proc. Network Operations and Management Symposium*, 2004.

[7] Clam AntiVirus. `http://www.clamav.net`.

[8] C. Dewes, A. Wichmann, and A. Feldmann. An Analysis Of Internet Chat Systems. In *Proc. ACM Internet Measurement Conference*, 2003.

[9] S. Dharmapurikar and V. Paxson. Robust TCP Stream Reassembly In the Presence of Adversaries. In *Proc. USENIX Security Symposium*, 2005.

[10] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational Experiences with High-Volume Network Intrusion Detection. In *Proceedings of ACM CCS*, 2004.

[11] DSniff. `www.monkey.org/~dugsong/dsniff`.

[12] J. Early, C. Brodley, and C. Rosenberg. Behavioral Authentication of Server Flows. In *Proc. Annual Computer Security Applications Conference*, 2003.

[13] ENDACE Measurement Systems. `http://www.endace.com`.

[14] Enterasys Networks, Inc. Enterasys Dragon. `http://www.enterasys.com/products/ids`.

[15] Freechal P2P. `http://www.freechal.com`.

[16] D. Fliegl, T. Baur, and H. Reiser. Nat-O-Mat: Ein generisches Intrusion Prevention System. In *Proc. 20. DFN-Arbeitstagung über Kommunikationsnetze*, 2006.

[17] P. Haffner, S. Sen, O. Spatscheck, and D. Wang. ACAS: Automated Construction of Application Signatures. In *Proc. ACM Workshop on Mining Network Data*, 2005.

[18] M. Handley, C. Kreibich, and V. Paxson. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Proc. 10th USENIX Security Symposium*, 2001.

[19] The Internet Corporation for Assigned Names and Numbers. `http://www.iana.org`.

[20] McAfee IntruShield Network IPS Appliances. `http://www.networkassociates.com`.

[21] The IPP2P project. `http://www.ipp2p.org/`.

[22] C. Kalt. Internet Relay Chat: Client Protocol. RFC 2812, 2000.

[23] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: Multilevel Traffic Classification in the Dark. In *Proc. ACM SIGCOMM*, 2005.

[24] Application Layer Packet Classifier for Linux. `http://l7-filter.sourceforge.net`.

[25] libmagic — Magic Number Recognition Library.

[26] M. Mai. Dynamic Protocol Analysis for Network Intrusion Detection Systems. Master's thesis, TU München, 2005.

[27] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proc. Winter USENIX Conference*, 1993.

[28] A. Moore and K. Papagiannaki. Toward the Accurate Identification of Network Applications. In *Proc. Passive and Active Measurement Workshop*, 2005.

[29] A. Moore and D. Zuev. Internet Traffic Classification Using Bayesian Analysis Techniques. In *Proc. ACM SIGMETRICS*, 2005.

[30] Linux NetFilter. `http://www.netfilter.org`.

[31] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[32] V. Paxson and S. Floyd. Wide-Area Traffic: The Failure of Poisson Modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–224, June 1995.

[33] T. Ptacek and T. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., 1998.

[34] M. Roesch. Snort: Lightweight Intrusion Detection for Networks. In *Proc. Systems Administration Conference*, 1999.

[35] M. Roughan, S. Sen, O. Spatscheck, and N. Duffield. Class of Service Mapping for QoS: A Statistical Signature Based Approach To IP Traffic Classification. In *Proc. ACM Internet Measurement Conference*, 2004.

[36] S. Sen, O. Spatscheck, and D. Wang. Accurate, Scalable In-Network Identification of P2P Traffic Using Application Signatures. In *Proc. World Wide Web Conference*, 2004.

[37] R. Sommer. *Viable Network Intrusion Detection in High-Performance Environments*. PhD thesis, TU München, 2005.

[38] R. Sommer and V. Paxson. Enhancing Byte-Level Network Intrusion Detection Signatures with Context. In *Proc. 10th ACM Conference on Computer and Communications Security*, 2003.

[39] K. Tan and B. Collie. Detection and classification of TCP/IP network services. In *Proc. Annual Computer Security Applications Conference*, 1997.

[40] K. Xu, Z.-L. Zhang, and S. Bhattacharyya. Profiling Internet Backbone Traffic: Behavior Models and Applications. In *Proc. ACM SIGCOMM*, 2005.

[41] Y. Zhang and V. Paxson. Detecting Backdoors. In *Proc. USENIX Security Symposium*, 2000.

Behavior-based Spyware Detection

Engin Kirda and Christopher Kruegel
Secure Systems Lab
Technical University Vienna
{ek,chris}@seclab.tuwien.ac.at

Greg Banks, Giovanni Vigna, and Richard A. Kemmerer
Department of Computer Science
University of California, Santa Barbara
{nomed,vigna,kemm}@cs.ucsb.edu

Abstract

Spyware is rapidly becoming a major security issue. Spyware programs are surreptitiously installed on a user's workstation to monitor his/her actions and gather private information about a user's behavior. Current anti-spyware tools operate in a way similar to traditional anti-virus tools, where signatures associated with known spyware programs are checked against newly-installed applications. Unfortunately, these techniques are very easy to evade by using simple obfuscation transformations.

This paper presents a novel technique for spyware detection that is based on the characterization of spyware-like behavior. The technique is tailored to a popular class of spyware applications that use Internet Explorer's Browser Helper Object (BHO) and toolbar interfaces to monitor a user's browsing behavior. Our technique uses a composition of static and dynamic analysis to determine whether the behavior of BHOs and toolbars in response to simulated browser events should be considered malicious. The evaluation of our technique on a representative set of spyware samples shows that it is possible to reliably identify malicious components using an abstract behavioral characterization.

Keywords: spyware, malware detection, static analysis, dynamic analysis.

1 Introduction

Spyware is rapidly becoming one of the major threats to the security of Internet users [20, 23]. A comprehensive analysis performed by Webroot (an anti-spyware software producer) and Earthlink (a major Internet Service Provider) showed that a large portion of Internet-connected computers is infected with spyware [1], and that, on average, each scanned host has 25 different spyware programs installed [7].

Different from other types of malware, such as viruses and worms, the goal of spyware is generally not to cause damage or to spread to other systems. Instead, spyware programs monitor the behavior of users and steal private information, such as keystrokes and browsing patterns. This information is then sent back to the spyware distributors and used as a basis for targeted advertisement (e.g., pop-up ads) or marketing analysis. Spyware programs can also "hijack" a user's browser and direct the unsuspecting user to web sites of the spyware's choosing. Finally, in addition to the violation of users' privacy, spyware programs are also responsible for the degradation of system performance because they are often poorly coded.

A number of anti-spyware products, whose goal is the identification and removal of unwanted spyware, have been developed. These tools are mostly based on the same technology used by anti-virus products. That is, they identify known spyware instances by comparing the binary image of these programs with a number of uniquely-characterizing signatures. These signatures are manually generated by analyzing existing samples of spyware. As a consequence, these anti-spyware tools suffer from the same drawbacks as signature-based anti-virus tools, including the need for continuous updating of their signature set and their inability to deal with simple obfuscation techniques [4].

This paper presents a novel spyware detection technique that overcomes some of the limitations of existing anti-spyware approaches. Our technique is based on an abstract characterization of the behavior of a popular class of spyware programs that relies on Internet Explorer's Browser Helper Object (BHO) and toolbar interfaces to monitor a user's browsing behavior. More precisely, our technique applies a composition of static and dynamic analysis to binary objects to determine if a component monitors users actions and reports its findings to an external entity. This characterization is independent of the particular binary image and therefore can be used to identify previously unseen spyware programs, and, in addition, it is resilient to obfuscation.

The main contributions of this paper are as follows:

- We introduce a novel characterization of the behavior of spyware components that are implemented as Browser Helper Objects or toolbars.

- We present novel static and dynamic analysis techniques to reliably identify malicious behavior in Browser Helper Objects and toolbar components.

- We present experimental results on a substantial body of spyware and benign samples that demonstrate the effectiveness of our approach.

The remainder of this paper is structured as follows. In Section 2, we present related work in the field of behavior-based malware detection in general and spyware detection in particular. Section 3 provides some background information on Browser Helper Objects and toolbars. It also shows how they are exploited by spyware programs to monitor user behavior and to hijack browser actions. In Section 4, we describe our abstract characterization of spyware-like behavior. In Section 5, we motivate the use of static and dynamic analysis for spyware detection, and we provide the details of our technique in sections 6 and 7. Section 8 discusses possible limitations of our proposed system. In, Section 9 we provide an experimental evaluation of the effectiveness of our technique. Finally, Section 10 briefly concludes and outlines future work.

2 Related Work

Spyware is difficult to define. There are many types of spyware that behave in different ways and perform actions that represent different levels of "maliciousness." For example, "adware" programs that present targeted advertisements to the user are considered less malicious than other forms of spyware, such as key-loggers, which record every single key pressed by the user. Regardless of the type of privacy violation performed, spyware is generally undesirable code that the user wants to remove from his/her system.

The increasing sensitivity of consumers to the spyware problem prompted a number of anti-spyware commercial products. For example, both AdAware [2] and SpyBot [22] are popular tools that are able to remove a large number of spyware programs. Recently, Microsoft released a beta version of an anti-spyware tool, aptly called Windows AntiSpyware [14].

Current spyware detection tools use signatures to detect known spyware, and, therefore, they suffer from the drawback of not being able to detect previously unseen malware instances. This is a deficiency shared by other malware-detection tools, such as anti-virus products and many network-based intrusion detection systems [17, 19]. Recently, researchers have tried to overcome these limitations by proposing behavior-based malware detection techniques. These techniques attempt to characterize a program's behavior in a way that is independent of its binary representation. By doing this, it is possible to detect entire classes of malware and to be resilient to obfuscation and polymorphism.

For example, in [5] the authors characterize different variations of worms by identifying semantically equivalent operations in the malware variants. Another approach is followed in [11], which characterizes the behavior of kernel-level rootkits. In this case, the authors use static analysis to determine if a loadable kernel module is accessing kernel memory locations that are typically used by rootkits (e.g., the system call table).

There are many advantages to representing malware in an abstract way. For instance, by using behavioral characterizations to detect malicious applications, one can obviate the need for a large data base of signatures to identify each known piece of malware. Another important benefit is that the characterization is resilient to malware variants and allows for the detection of previously unseen malware instances. An example of using behavior characterization is Microsoft's Strider Gatekeeper [24]. This tool monitors *auto-start extensibility points* (ASEPs) to determine if software that will be executed automatically at startup is being surreptitiously installed on a system.

Our approach is similar to the one pursued by Strider Gatekeeper, because our goal is to model spyware-like behavior. Consequently, our behavioral classifications are not specifically tailored to a single spyware program, but, instead, they are able to detect entire classes of spyware applications. However, our technique is more powerful than the one used by Strider Gatekeeper, because it identifies a more general behavior pattern, which is the acquisition of private information and the leaking of this information outside the boundaries of the application. In addition, our technique uses a combination of static and dynamic analysis, which allows for a very precise characterization of the behavior of an application in reaction to browser events, thus, reducing the chance of false positives.

The use of static analysis to analyze the behavior of malware has been proposed previously in [5]. However, the technique proposed by Christodorescu et al. is tailored to detect different variations of the same malware (e.g., different versions of the NetSky worm). Our technique, instead, focuses on abstract, spyware-like behavior and is not limited to detecting just one spyware program and its variants.

The technique presented in this paper, however, is not completely general. We focus explicitly on one type of spyware, that is, malware that exploits the hooks provided by Microsoft's Internet Explorer to monitor the actions of a user. This is done by using the Browser Helper Object (BHO) interface or by acting as a browser toolbar object. Our initial focus is justified by the fact that the overwhelming majority of spyware has a component based on one of these two technologies. This is confirmed not only by our own experience in analyzing various spyware components but also by a recent study [24], which found that out of 120 distinct spyware programs, just under 90 used BHOs as an entry point to monitor user activity and approximately 46 used the IE toolbar mechanism (note that some spyware programs used both mechanisms). In addition to these findings, a US-CERT report [8] names BHOs as one of the more frequently used techniques employed by spyware along with browser session hijacking and stand-alone applications. Other forms of spyware (e.g., stand-alone applications) are not currently being addressed but are open questions for future work. Additionally, we recognize that spyware does not only affect the Microsoft Windows platform or the Internet Explorer browser. Mozilla products also have their problems [15], and have not been completely free from spyware [9, 26]. Other platforms also contain ASEP hooks similar to those found in Windows as described in [24]. However, as of today the occurrence of spyware affecting other platforms and browsers is significantly lower than that affecting both Microsoft Windows and Internet Explorer. Still, the growing popularity of alternatives make this an important consideration for the future.

Because of the relevance of the Component Object Model (COM) architecture and the hooks that Internet Explorer provides, the next section presents some background material to help the reader who is not familiar with these concepts. The following sections then present the details of our detection technique.

3 Spyware, Browser Helper Objects, and Toolbars

Spyware authors have many options on a Windows host when it comes to looking for good vantage points from which to glean personal information about users. For example, Layered Service Providers, which sit between the application-level network APIs (Application Programming Interfaces) and the kernel, can filter network traffic and/or collect information about users. Another example is represented by background processes that are automatically executed at startup to monitor user actions. Because sensitive information is often accessed through web-based interfaces, browser plug-ins are another popular mechanism to collect sensitive data and monitor user actions. Internet Explorer plug-ins, and in particular Browser Helper Objects (BHOs) and toolbars, as mentioned previously, are used in the majority of spyware programs as a mechanism to access information about a user's browsing habits or to control the browser's behavior.

Browser Helper Objects and toolbars are binary objects that conform to the Component Object Model (COM). COM is a binary standard developed by Microsoft to support, among other things, a component-based software market [25]. Every COM object implements a set of interfaces, each of which is a well-defined contract that describes what functionality the object provides. The COM standard guarantees that the virtual tables of interfaces remain the same across compilers, allowing COM objects to be implemented and used by any language that supports calling functions through a table of function pointers. The `IUnknown` interface must be implemented by all COM objects. It contains reference-counting functionality and the function `QueryInterface`, which allows one to query for the other interfaces that an object might implement.

A Browser Helper Object is in essence a simple COM object that implements the `IObjectWithSite` interface. Toolbar objects work in a way similar to BHOs and, in addition, implement a few more interfaces and include a graphical component.

At startup, Internet Explorer loads all the BHOs that are registered as COM servers and whose Class Identifiers (CLSIDs) are included under the registry key `\HKLM\SOFTWARE\Windows\CurrentVersion\Explorer\Browser Helper Objects`. Toolbars are loaded in a similar fashion with their CLSIDs being present under several other keys. Then, for each loaded BHO or toolbar, the browser calls the `SetSite` method exported by the `IObjectWithSite` interface, passing a reference to the browser's `IUnknown` interface as a parameter. This reference can then be used by a BHO or toolbar to query for other interfaces implemented by the browser. Interfaces of interest include `IWebBrowser2`, which allows a BHO or toolbar to access the current document and Uniform Resource Locator (URL) as well as to load specific pages, and `IConnectionPoint`, which allows a BHO to monitor the browser events specified in `DWebBrowserEvents2`. Toolbar objects access browser interfaces in a similar way.

By invoking the methods provided by the browser interfaces, BHOs and toolbars can completely control the browser's behavior and access sensitive data entered by the user during navigation. Because of this, BHOs and toolbars are often used as the core components in spyware applications.

4 Spyware Characterization

A distinctive characteristic of spyware is the fact that a spyware component (or process) collects data about user behavior and forwards this information to a third party. Since we restrict our focus to spyware that is implemented as a BHO or toolbar, we have to identify the mechanisms that these components employ **(i)** to monitor user behavior and **(ii)** to leak the gathered data to the attacker.

The most straightforward mechanism to monitor user behavior is to subscribe to browser events (using the browser's `IConnectionPoint` interface). When subscribing to these events, a component is notified in response to almost all browser actions or state changes. For example, events are generated when a new URL is opened, when a requested page cannot be found, or when the download of a resource has completed. In response to an event, the browser extension can request more information using the COM interfaces offered by the browser. In particular, the BHO or toolbar can request a handle to the document that was accessed, so that it can perform further analysis. In addition, a BHO or toolbar can react to an event by directing the browser to another page or opening a pop-up window. In summary, we expect to see the spyware component interact with the web browser by invoking browser functions in response to events.

After the spyware component has extracted the desired information, its next task is to transmit the data to a third party (the attacker). To this end, the information must be either directly transfered over the network, passed to a cooperating process running on the same host, or stored locally (for example, in a file on disk or in the Windows Registry). In any case, the spyware component has to interact with the operating system to be able to deliver the data to the intended recipient. [1] Even when data is temporarily kept in memory, eventually an operating system service has to be invoked to leak the data.

Because the interaction with the operating system is necessary for a spyware component, we analyze the operating system services that the component requests. In particular, we focus on the Windows API [2] calls that a component can use to leak information from the current process. Again, we are only interested in the Windows API calls that are performed in response to events. The reason is that for a Browser Helper Object, event handling code is the only code that is executed after the component's startup phase. With a toolbar, code can also be executed when the user clicks on a user interface element belonging to the toolbar. However, there is the significant risk that the web browser will be closed without ever making use of the toolbar. In this case, all information would be lost because the spyware code is simply not run. Thus, information can only be reliably leaked in event handling code (the last chance at which this is possible being in response to the event that signals the browser is being closed).

There is one important exception to the assumption above: A spyware component could attempt to set up an additional path through which events can be leaked by starting another thread or by registering a timer with an appropriate callback function. In response to a browser event, the collected user information would not be leaked immediately, but it would first be stored in a globally accessible data structure. Later, the second thread or the timer callback function could flush out this information without being detected. To eliminate this possibility for evading detection, we chose a conservative approach. That is, whenever a component can create either a thread or a timer, *all* Windows API calls that this component can invoke are included in the analysis, not only those invoked in response to events.

We realize, of course, that benign components may interact with the web browser in response to events to provide some service to the user, such as automatically displaying the page source in a separate frame. Benign components may also interact with the operating system (via Windows API calls). For example, a component could read some configuration parameters from a file, write entries to a log-file, or download updates to the software at startup. The key insight here is that each of the two characteristics on their own does not generally warrant suspicion, but together they are strong indicators of malicious behavior. Therefore, we classify a browser helper object or a toolbar as spyware if the component, in response to browser events,

1. monitors user behavior by interacting with the web browser **and**
2. invokes Windows API calls that can potentially leak information about this behavior (e.g., calls to save the data to a file or transmit information to a remote host).

Note that our classification is more general than the one used by virus scanners and signature-based intrusion detection systems, as we are looking for intrinsic behavioral characteristics of spyware instead of byte strings specific to particular malware instances.

Our detection approach can take advantage of the proposed spyware classification in one of two ways.

[1]Note that we are not taking into account the possibility of using covert channels to leak the information from the BHO or toolbar to the outside environment. Typically, these channels are either bandwidth-limited or generate distinctive patterns of execution.

[2]The Windows API provides a large number of procedures that can be invoked to access the complete functionality of the Windows operating system (this includes kernel services, the graphical device interfaces, and other user interfaces).

First, we can compare the results of our analysis (that is, the identification of BHO/toolbar-to-browser and BHO/toolbar-to-OS interactions in response to events) to an *a priori* assembled list of browser COM functions and Windows API calls that we deem malicious. In this way, we define our behavioral characterization based on prior experience and use this characterization to detect previously unknown instances of spyware. The second method is to automatically generate a behavioral characterization by comparing the behavior of known benign components to the behavior of known malicious ones. More precisely, the characterization is automatically derived by identifying the browser functions and Windows API calls performed by malicious BHOs and toolbars that are not also executed by benign samples. This characterization would then be used to identify previously unseen spyware samples.

In the following, we apply the first approach described above. We then use the second approach as a way to validate our choice of characterizing browser functions and Windows API calls.

5 Component Analysis

Given our characterization of spyware, the task of the analysis phase is to extract the behavior of an unknown BHO or toolbar. That is, we are interested in the interaction of an unknown sample with the browser and with the operating system in response to browser events. Based on the results of the analysis, we can then classify the sample appropriately.

As a first step, we propose a dynamic analysis technique that exposes a suspicious component to crafted browser events (which simulate user activity) and analyzes the component's response. In particular, we dynamically record both the browser COM functions and the Windows API functions that the component calls. This approach exhibits some commonalities with black-box testing in that data is sent to a component under examination and its behavior is analyzed, without knowing anything about the component's implementation.

An interesting problem is to determine events and input that are suitable to capture the behavior of a component. When performing black-box testing, it is typically a significant challenge to devise a set of test inputs that exercise the bulk of the functionality (or code paths) of the object under test. In our case, the situation is exacerbated by the fact that we have no *a priori* knowledge about the functionality of the (potentially malicious) component. Thus, it may be difficult to generate input that will reveal spyware-like behavior with a sufficiently high probability. Consider, for example, a spyware component that scans all web pages that are fetched by the browser for the occurrence of certain keywords (e.g., "car insurance"). The corresponding URL is logged to a file *only* when these keywords appear on the page. Thus, we would observe suspicious behavior (in the form of a file system call) only if one of the test pages actually contains the words "car insurance." Another problem is that spyware often does not react consistently to identical events. For example, a spyware developer might decide that a user would be exceedingly annoyed if a pop-up window containing an advertisement appeared every time a page with a specific keyword was accessed. Therefore, the pop-up would be opened only occasionally, and, as a consequence dynamic black-box testing alone might not be enough to observe all interesting reactions.

Before discussing appropriate extensions to our analysis, an important point in the previous discussion is the fact that the lack of coverage in the dynamic test affects (almost) exclusively the interaction of the component with the operating system and not its interaction with the web browser. The reason is that a component has to monitor the user behavior by interacting with the web browser *before* any decision can be made. For example, before a page can be scanned for the occurrence of a certain keyword, the spyware component has to first request the page source from the browser. Also, a spyware component might not necessarily log all URLs that a user visits. However, before the decision to write the URL to a log-file can be made, the visited URL must be retrieved from the browser. A spyware component might decide to record only certain pieces of information or engage with the environment only under certain circumstances. However, as a first step, it is always necessary to extract information about the current document from the web browser. Thus, the results of the dynamic analysis with regards to the interaction with the web browser are sufficiently comprehensive.

To capture all possible reactions of a component with regards to its interaction with the operating system, we complement the dynamic testing with a static analysis step. To be more precise, we use dynamic analysis to locate the entry point into the code of the component that is responsible for handling events (i.e., the object's `Invoke` function). Starting from this entry point, the static analysis step extracts the control flow graph of all code regions that are responsible for handling events. The key observation is that this control flow graph contains (or encodes) all possible reactions of the component to events. As a consequence, there is no need to confine the analysis to the API calls that are actually observed during dynamic testing. Instead, we can analyze all Windows API calls that this component can possibly invoke when receiving events. Following up on the example of the spyware that scans for "car insurance" in fetched pages, we extract the control flow graph of the code that handles a "web page received" event during our analysis and determine that the component might perform a file system access under certain circumstances. Note that it is necessary to confine the static analysis to those code regions that are responsible for handling events. Otherwise, the analysis could end up including API function invocations that cannot be used for leaking information (e.g., API calls during program startup). Taking into account these additional function calls could lead to incorrect classification results.

As static analysis is already required to identify the interaction of a component with the operating system, one might consider dropping the dynamic analysis step. That is, one could attempt to solely rely on static analysis to recover the interaction of a component both with the web browser and with the operating system. This is difficult for a number of reasons. One problem is that Browser Helper Objects or toolbars often contain a number of different COM objects. Usually, it is not *a priori* evident which of these objects will register for browser events. Even when the correct COM object can be identified, locating the code regions that are responsible for handling events is not straightforward. Finally, COM function invocations are implemented as indirect function calls (via the COM object's virtual function table). Thus, it would be significantly more difficult to statically recover the interaction of the BHO or toolbar with the web browser.

To summarize, both static and dynamic analysis techniques have their advantages. The dynamic analysis step can precisely pinpoint the COM object and the code regions that handle browser events. In addition, the interactions between a component and the web browser can be accurately captured. Static analysis, on the other hand, is more comprehensive in identifying possible interaction of a component with the operating system. Thus, we believe that a combination of dynamic and static analysis is most suitable to analyze the behavior of unknown browser helper objects and toolbars.

In the previous discussion, we have not considered the fact that a spyware component might react differently to different events. That is, the dynamic analysis step produces a single list of COM browser functions that are called by the component under test, and the static analysis step yields a single list of Windows API calls that can be reached in response to events. However, it might be beneficial to distinguish between responses associated with different events. For example, a component might perform a suspicious Windows API call only in response to a certain event that conveys no relevant information about a user's actions (e.g., an event that signals that the browser window has been resized). In this case, the suspicious API call may not leak any information, and excluding it from the list could reduce false positives. Unfortunately, restricting automatic analysis to certain relevant events offers spyware a way to evade detection. To see this, consider a spyware component that monitors user behavior (via browser COM functions) in response to events, but, instead of immediately leaking this information, stores it temporarily in memory. Later, the collected information is flushed in response to an event that is not considered by our analysis. Therefore, we decide to take the more conservative approach in order to be more resilient to evasion.

As was mentioned in the previous section, our technique makes use of a more general characterization of malware behavior than that used by current signature-based systems. However, the methods used to realize this detection strategy are not without their drawbacks. While both our tool and the signature-based systems can be classified as detection engines, the latter are much faster and are practical for common desktop usage, while our technique is more appropriate for analysts who are looking to classify unknown instances of malware in an automated fashion.

In the following sections, we explain in detail the dynamic and static analysis steps that we perform to determine behavioral characterizations for spyware. We then discuss possible methods of evasion for our technique. Finally, we present experimental data that demonstrates that our detection technique is capable of effectively distinguishing malicious and benign components.

6 Dynamic Analysis Step

The basis for our detection technique, as discussed in the previous sections, is to extract behavioral characterizations based on how spyware interacts with **(i)** the browser and **(ii)** the underlying operating system in response to events. To this end, we make use of both dynamic and static analysis techniques.

The goal of the dynamic analysis step is twofold. First, it has to monitor the interaction of the component with the browser and record all the browser's COM functions that are invoked in response to events. Second, it has to determine the code regions that are responsible for handling events, thereby providing the necessary starting points for the static analysis step. These tasks are accomplished with the help of three core elements.

The first element is a "fake" WebBrowser COM object, which provides the component under analysis with an environment similar to the one that would be present when being hosted by an instance of Internet Explorer. The second element is our COM object host application, which properly instantiates all involved components and sends the relevant browser events to the BHO or toolbar component under evaluation. The final element is a program that traces the execution of our host application to extract those code regions that handle the various browser events that are delivered.

6.1 Recording Browser Function Calls

The "fake" WebBrowser and the host application provide a controlled environment in which we can instantiate a suspicious component, send events, and monitor the component's reaction. The purpose of the "fake" WebBrowser COM object is to host the component under analysis. This involves the provision of an environment that is "convincing" to a BHO or toolbar. To this end, the WebBrowser element must offer all Internet Explorer functionality expected by the browser extensions. Otherwise, a BHO or toolbar might fail during initialization, preventing any further analysis. Therefore, the WebBrowser COM object implements several key interfaces expected by a BHO or toolbar. Most importantly, it implements `IConnectionPointContainer`, `IConnectionPoint`, and `IWebBrowser2`. The two interfaces related to `ConnectionPoint` are required so that a BHO or toolbar is able to notify our WebBrowser COM object about its interest in receiving events (using the `IConnectionPoint::Advise()` function). The `IWebBrowser2` interface is the main interface used to interact with the browser. More precisely, a browser component invokes the functions of this interface to collect browsing information such as the current page or URL and to influence browser behavior. Since we are interested in the interaction between a possible spyware component and the browser, we pay particular attention to calls to the `IWebBrowser2` interface.

The WebBrowser element also implements several other interfaces that are expected by a toolbar. These include `IOleWindow`, `IInputObjectSite`, `IOleCommandTarget`, and `IServiceProvider`. Note that implementing an interface does not necessarily imply that it is necessary to faithfully simulate the functionality of all its procedures. Instead, we usually provide a stub for every function. This stub always returns success and logs the invocation. However, for certain frequently-used functions (e.g., those that request the document or the location of the current page), appropriate objects are returned.

The host application provides the "glue" that will hold the various components together. To this end, the program registers the component under analysis with the Windows operating system, initializes the COM library, and instantiates both our WebBrowser and the BHO or toolbar.

Before events can be sent to a component, its `SetSite` method has to be called (with a pointer to the `IUnknown` interface of our WebBrowser as an argument). If the component is actually interested in receiving events, it will respond by querying for the WebBrowser's `IConnectionPoint` interface and call its `Advise` function. At his point, the host application can obtain a reference to the `IDispatch` interface implemented by the browser extension and start to send events. Note that all events are delivered through a single function (the `Invoke` method of the `IDispatch` interface), using the first parameter of the function to indicate the type of the event.

To be able to generate events that are as realistic as possible, we recorded an actual event stream created by Internet Explorer over the course of five days. To this end, we developed our own BHO that logged all relevant event information while the browser was used. These events were then replayed to perform our dynamic tests.

6.2 Locating Event-Handling Code

Given the infrastructure to send browser events to a Browser Helper Object or toolbar, the next task is to determine the regions of the code that handle events. Moreover, we are interested in determining the *separate* code regions associated with each event. Then, we can use static analysis to extract the control flow graphs that correspond to these events.

If each different type of event would be passed to a separate function, the start address for the static analysis process could be easily determined as the start address of the respective function. Unfortunately, as mentioned in the previous section, all events are delivered through the single `Invoke` function. Thus, if we were to use the start address of the `Invoke` function, we would be unable to determine which API calls are associated with which events. To obtain the API calls made in response to a specific event, we have to look deeper into the component under analysis and find the first instruction in the code that is responsible for handling the event. We call this instruction the first *event-specific instruction*. Of course, it is also possible that a component is not interested in a certain event and provides no special handling code. The execution typically runs through a default path, ignoring the information contained in the event. In this case, there is no event-specific code.

To determine the first event-specific instructions, we collect execution traces of the BHO or toolbar when processing different events. That is, we send one event of each type to the component and record the corresponding sequences of machine instructions that are executed in response. To record the machine instruction traces, we use the Windows Debug API [21]. The Windows Debug API offers an interface that is comparable to the ptrace mechanism provided by some UNIX implementations, and it provides a parent process with complete control over the execution of a child process. This includes the possibility to read and write the registers as well the address space of the child process, which allows one to set breakpoints or run a process in single-step mode. By switching to single-step mode before sending a browser event, we can record each executed machine instruction and obtain the desired traces.

The tracing component collects an execution trace for each of the n events being analyzed. Then the application performs a pairwise comparison between all traces. The idea is to identify the first event-specific instruction for each event by checking for the first instruction that is unique to the corresponding trace. More formally, a trace t_e for an event e can be considered as a string whose symbols are the (addresses of the) instructions that are executed. To identify the first event-specific instruction in t_e, we determine the longest common prefixes between t_e and all other traces $t_i : 0 \leq i < n, e \neq i$. Assuming that the longest of the prefixes has a length of l, then the $(l+1)^{\text{st}}$ instruction in trace t_e is the first event-specific one. The rationale behind this approach is that we search for the first instruction for which the trace t_e deviates from *all* other traces. As a consequence, when two or more traces contain the same sequence of instructions, then these traces have no event-specific instructions and are considered to represent the default path (as the different types of events had no influence on the execution).

Consider the example shown in Figure 1. Note that in this case, we demonstrate the identification of event-specific instructions using source code. However, the real analysis is done on binary code. The figure shows the traces generated for five events. As expected, the correct event-specific instructions are found for the first three events (line 4 for event A, line 6 for event B, and line 8 for event C), while the last two events (D and E) represent the default path. Note that even though the first instruction in trace B that is different from trace A is on line 5, there is a longer common sequence of this trace with trace C (as well as D and E). Thus, the event-specific instruction for event B is determined to be on line 6.

When collecting traces for Browser Helper Objects or toolbars, only the instructions that are executed in the context of the component itself are used to determine event-specific instructions. Thus, we remove all instructions that belong to dynamically loaded libraries from the traces. The reasons are twofold: First, we are interested in finding the first unique instruction *within the component* for the static analysis process. Second, a library can contain initialization code that is executed when one of its functions is used for the first time. This introduces spurious deviations into the traces that do not correspond to actual differences in the code executed by the BHO or toolbar component.

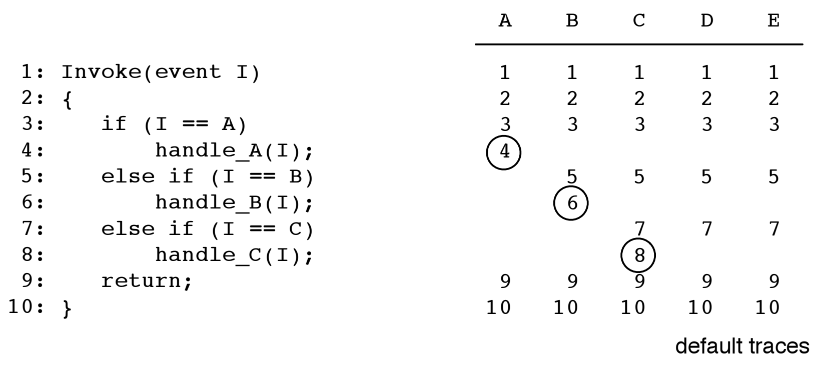
In addition to restricting our analysis to code within the BHO or toolbar, subsequent repetitions of identical instruction sequences that are executed as part of a loop are collapsed into a single instance of this sequence. The reason is that we occasionally observed that the traces for two events were identical before and after a loop, while the loop itself was executed for a different number of times in each case. This happened, for example, with a spyware component that was going through an array of identifiers to determine whether the current event (given its identifier) should be processed. For different event identifiers, the loop terminated after a different number of iterations because the respective event identifiers were found at different positions in the array. However, for both traces execution continued on the same path for a number of instructions until control flow eventually branched into the event-specific parts. In such situations, collapsing multiple loop iterations into one allows us to identify the actual event-specific handling code.

7 Static Analysis Step

The goal of the static analysis step is to determine the interaction of a BHO or toolbar component with the operating system. To this end, we statically examine certain code regions of a component for the occurrence of operating system calls.

Before the component is actually analyzed, we check its API function import table for the occurrence of calls relevant to the creation of threads or timers (e.g., `CreateThread` or `SetTimer`). As explained in Section 4, if a component could launch threads or create timers, we have to conservatively assume that any imported API function can be invoked in response to an event. In this case, no further analysis of the binary is necessary because we can directly use the calls listed by the function import table. When neither of these functions is present, however, the static analysis step is required to identify those API functions that can be called in response to events.

The first task of the static analysis step is to disassemble the target binary and generate a control flow graph from the disassembled code. A control flow graph (CFG) is defined as a directed graph $G = (V, E)$ in which vertices $u, v \in V$ represent basic blocks and an edge $e \in E : u \rightarrow v$ represents a possible flow of control from u to v. A basic block describes a sequence of instructions without any jumps or jump targets in the middle. We use IDA Pro [6] to disassemble the binary. Since IDA Pro is a powerful disassembler that already provides comprehensive information about the targets of control flow instructions, the CFG can be generated in a straightforward manner using a custom-written IDA Pro plug-in. Note that if our detection technique were to be deployed in



```
                                          A    B    C    D    E
                                         ------------------------
 1: Invoke(event I)                       1    1    1    1    1
 2: {                                     2    2    2    2    2
 3:    if (I == A)                        3    3    3    3    3
 4:        handle_A(I);                   4
 5:    else if (I == B)                        5    5    5    5
 6:        handle_B(I);                        6
 7:    else if (I == C)                             7    7    7
 8:        handle_C(I);                             8
 9:    return;                            9    9    9    9    9
10: }                                    10   10   10   10   10
```

Figure 1: Dynamic traces for different types of events.

the general public, the disassembly and CFG generation would be done using a custom disassembler optimized for our task. During our experiments, we encountered a number of spyware samples that were compressed with UPX [16], a packer tool for executables. If this was the case, we uncompressed the samples prior to performing static analysis (using the available UPX unpacking utility). Otherwise, IDA Pro would not be able to extract any valid instructions.

Based on the CFG for the entire component, the next step is to isolate those parts of the graph that are responsible for handling events. In particular, we are interested in all subgraphs of the CFG that contain the code to handle the different events. To this end, we use the event-specific addresses collected during dynamic analysis and traverse the entire subgraph reachable from each of those addresses. While traversing the graph, the static analysis process inspects all instructions to identify those that represent operating system calls. More specifically, we make a list of all possible Windows API calls that can be reached from each event-specific address. Finally, the event specific lists are merged to obtain a list of all API calls that are invoked in response to events. At this point, the analysis process has collected all the information necessary to characterize the component (i.e., browser COM functions and Windows API calls executed in response to events).

Note that while the Windows API is the common way to invoke Microsoft Windows services, current versions of Windows (starting with Windows NT and its successors) also offer a lower-level interface. This interface is called the Windows NT Native API, and it can be compared to the system call interface on UNIX systems. Both the Native API kernel interface and the Windows API are offered to accommodate the micro-kernel architecture of Windows. That is, instead of providing one single operating system interface, Windows NT offers several different operating system interfaces (e.g., OS/2, DOS, POSIX). This allows one to execute applications that were developed for different operating systems. The different OS interfaces are implemented by different operating environment subsystems, which are essentially a set of system-specific APIs implemented as DLLs that are exported to client programs. All subsystems are layered on top of the Native API, with the Windows API being the most popular subsystem. Because applications typically use the Windows API and not the Native API, we monitor calls to the Windows API to capture the behavior of components under analysis. However, to assure that no spyware can bypass our detection technique by relying directly on the Native API, any direct access to this interface is automatically characterized as suspicious.

8 Evading Detection

In this section, we discuss the limitations of our detection technique. In particular, we explore possible mechanisms that a spyware author can use to evade detection and countermeasures that can be taken in order to prevent such evasion.

Before revisiting our technique, it is important to note that due to the nature of the component object model a component that "plugs" into *our* WebBrowser component must use the interfaces it exposes (or the ones it expects Internet Explorer to expose) in order to extract information about the user at runtime. These interfaces are well documented and are essentially a contract between the COM client and the COM server. Since we control our WebBrowser component, we see all the interface calls and the queries for different interfaces. Just as our *a priori* list of suspicious Windows API calls is subject to change as we discover new suspicious calls, so are the various COM functions and interfaces used. This contract also applies to events. If a component wants to receive events from the WebBrowser it *must* call the `Advise` function on the WebBrowser's `IConnectionPoint` interface. Since we are in control of our WebBrowser object we can monitor calls to this interface and reliably discover if a component is interested in events and, if it is, the address of the function that handles those events.

Recalling our behavioral characterization of spyware, we note that a BHO or toolbar component must both collect user data via browser functions and leak this information to the adversary via Windows API calls. Thus, to evade detection, a malware author could either attempt to hide the fact that the BHO monitors user data via browser COM calls, or disguise the fact that the collected data is leaked.

Covering the footprints that indicate user data is being collected is likely the more difficult task. We use dynamic analysis to monitor all the functions that the BHO component invokes in our web browser. To avoid invoking the browser functions, a spyware component could attempt to read interesting user data directly from memory. This is possible because both the BHO and the web browser share the same address space. However, this is difficult because a non-standard access to memory regions in a complex and undocumented COM application, such as Internet Explorer, is not likely to yield a robust or portable monitoring mechanism. Thus, reading data directly from memory is not considered to be a viable approach.

A more promising venue for a spyware component to evade our detection is to attempt to conceal the fact that data is leaked to a third-party via API calls. We have previously mentioned the possible existence of covert channels, and concluded that their treatment is outside the scope of this paper. However, a spyware component could attempt to leak information using means other than API calls, or it could prevent the static analysis process from finding their invocations in the code of the BHO.

One possible way to leak information without using the Windows API is to make use of the functionality offered by Internet Explorer itself. For example, a spyware component could use the Internet Explorer API to request a web resource on a server under the control of the attacker. Sensitive information could be transmitted as a parameter of the URL in this request. The current limitation of not taking browser calls into account can be addressed in two ways. First, we could extend the static analysis step to also flag certain COM calls to the browser as suspicious. The problem with this solution is that COM calls are invoked via function pointers and, thus, are not easily resolvable statically. The second possibility would be to extend the dynamic analysis step. We already record the browser functions that a BHO invokes to determine when user data could be leaked. Thus, it would be straightforward to additionally take into consideration browser calls that a component invokes after user data has been requested. However, for this, one has to enlarge the set of test inputs used for the dynamic analysis step to ensure better test coverage.

As mentioned previously, another evasion venue is to craft the BHO code such that it can resist static analysis. Static analysis can be frustrated by employing anti-disassembling mechanisms [13], or code obfuscation. If these techniques are used, then our static analysis step could be forced into missing critical Windows API calls that must be recognized as suspicious. Again, we have two options to deal with this problem. First, the static analysis step could be made more robust to tolerate obfuscation (e.g., by using a disassembler that handles anti-disassembler transformations [12]). Also, strong obfuscation typically leads to disassembly errors that in itself can be taken as sufficient evidence to classify a component as spyware. A second approach is to expand the dynamic analysis step to also monitor Windows API functions. This could be achieved by hooking interesting API calls [10] before the spyware component is executed. Using these hooks, all Windows API calls made by the spyware component could be observed. Again, the set of test data would have to be enlarged to improve test coverage.

While there are a number of possible ways that a spyware component could attempt to evade our current detection system, we have shown how to counter these threats. Furthermore, in the next section we show how in its current form our system was successful in correctly identifying all spyware components that we were able to collect. Thus, our proposed techniques significantly raise the bar for spyware authors with respect to traditional signature based techniques.

9 Evaluation

In order to verify the effectiveness of our behavior-based spyware detection technique, we analyzed a total of 51 samples (33 malicious and 18 benign); 34 of them were BHOs and 17 were toolbars. The process of collecting these samples in the "wild" is both a tedious and non-trivial task. This is confirmed by a recent study [3] in which the authors traversed 18,237,103 URLs discovering 21,200 executables, of which there were just 82 unique instances of spyware as identified by popular spyware scanners. The problem is further exacerbated by the fact that popular spyware dominates the set of infected files, making it hard to obtain a well rounded collection. Thus, we obtained all of the malicious samples in our final test set from an anti-virus company and collected all of the benign samples from various shareware download sites. Note that we picked *all* samples (benign and malicious) that we collected and that either registered themselves as BHOs or as toolbars. While collecting the benign samples, we verified that the applications were indeed benign by checking both anti-spyware vendor and software review web sites. Furthermore, we selected samples from different application areas, including anti-spyware utilities, automated form-fillers, search toolbars, and privacy protectors. Note that our tool was developed while analyzing only seven (two benign and five malicious) samples from our final test set. The remaining samples were effectively unknown, with respect to our tool, thereby validating the effectiveness of our characterization on new and previously unseen malware components. Given the difficulty of collecting samples, we consider this to be a well rounded and significant sample set with which to evaluate our technique.

Table 1 presents our detection results in terms of both correctly and incorrectly classified samples. In addition to the detection results for our proposed combined approach, this table also includes the results that are achievable when taking into account the information provided by only the static analysis or only the dynamic analysis step. In particular, we show detection results when the classification is solely based on statically analyzing *all* API calls invoked by the sample (Strategy 1) or only those API calls in response to events (Strategy 2). Moreover, we present the results obtained when a BHO or toolbar sample is classified as spyware if it subscribes to browser events (Strategy 3) or solely based on its interaction with the browser via COM functions (Strategy 4). Finally, Strategy 5 implements our proposed detection technique, which uses a composition of static and dynamic analysis. The aim is to demonstrate that the combined analysis is indeed necessary to achieve the best results.

Given our detection results, it can be seen that malicious spyware samples are correctly classified by all five strategies, even the most simple one. Since every strategy focuses on the identification of one behavioral aspect present in our characterization of spyware, these results indicate that the proposed characterization appears to accurately reflect the actual functioning of spyware. However, simple strategies also raise a significant number of false alarms. The reason is that certain behavioral aspects of spyware are also exhibited by benign samples. In the following paragraphs, we discuss in more detail why different detection strategies incorrectly classify certain samples as malicious. The discussion sheds some light on the shortcomings of individual strategies and motivates the usage of all available detection features.

As mentioned in Section 4, we need a list of Windows API calls that contains all suspicious functions that can be used by a spyware component to leak information to the attacker. As a first step, we manually assembled this list by going through the Windows API calls, in particular focusing on functions responsible for handling network I/O, file system access, process control, and the Windows registry. Figure 2 shows an excerpt of the 59 suspicious calls that were selected. The calls that are depicted are representative of commonly used registry, file access, and networking functions.

The first detection strategy (Strategy 1) uses the list of suspicious API functions to statically detect spyware. To this end, static analysis is used to extract *all* API calls that a sample could invoke, independent of events. This can be done in a straightforward fashion, using available tools such as PEDump [18]. Then, the extracted API calls are compared to the list of suspicious functions. A sample is classified as spyware if one or more of the sample's API calls are considered suspicious.

Using the first strategy, *all* benign samples are incorrectly detected as spyware. In many cases, samples require Windows registry, file, or network access during the startup and initialization phase. In other cases, benign samples such as the Google search toolbar use suspicious calls such as `InternetConnectA` to connect to the Internet (in the case of the Google toolbar, the sample sends search queries to Google). However, such calls are typically not done in response to events; in fact, many samples do not even register for browser events.

If we restrict the static analysis to only those Windows API calls that are invoked *in response* to browser events, only five of the 18 benign samples are incorrectly classified (Strategy 2). Two of these false positives are easy to explain. One is a BHO called `Airoboform`, a tool that supports users by filling in web forms automatically. In response to every event that signals that a new page is loaded, this tool scans the page for web forms. If necessary, it loads previously provided content from a file to fill in forms or it stores the current form content to this file. Because web forms can also contain sensitive information (such as passwords), one can argue that `Airoboform` actually behaves in a way that is very similar to a spyware application. The only exception is that in the case of spyware, the file content would probably be transmitted to an attacker through an additional helper process.

Besides `Airoboform`, the benign `Privacy Preferences Project (P3P) Client` BHO also exhibits spyware-like behavior. P3P is emerging as an industry standard for providing a simple and automated way for users to control the use of their personal information on web sites they visit. To this end, the `P3P Client` has to check the P3P settings of every web page that is visited. More precisely, whenever the user visits a web site, the BHO connects to that site and tries to retrieve its P3P-specific privacy policy. This is implemented by opening a connection via the Windows API function `InternetConnectA` in response to the event that indicates that a document has been loaded.

Detection Strategy	Spyware Components		Benign Components	
	Correct	Incorrect	Correct	Incorrect
1. All Windows API calls (static)	33	0	0	18
2. Windows API calls in response to events (static)	33	0	13	5
3. Subscription to browser events (dynamic)	33	0	10	8
4. Browser COM method invocations (dynamic)	33	0	15	3
5. Combined static and dynamic analysis	33	0	16	2

Table 1: Results for different detection strategies.

```
CreateFileA
DeleteFileA
WriteFile
fopen
fwrite
fopen

InternetConnectA
InternetOpenA
InternetReadFile
InternetConnectA
InternetOpenA
InternetOpenUrlA
InternetReadFile
InternetSetCookieA
WSACleanup
WSAStartup

RegCreateKeyExA
RegDeleteKeyA
RegDeleteValueA
RegSetValueExA
```

Figure 2: Excerpt of *a priori* assembled list of suspicious Windows API calls.

Two other false positives are `Spybot` and the `T-Online` toolbar. In both cases, the static analysis results indicate that a suspicious `WriteFile` call could be invoked in response to some events. This would allow the browser extensions to write event-specific information into a file for later retrieval. Although writes to a file are generally suspicious in response to events, there are also cases in which such an action is legitimate. For example, we discovered that the `T-Online` toolbar, a benign application that allows users to send SMS messages, uses a caching mechanism to store images in files. `Spybot`, a benign anti-spyware application, uses black lists to block web access to spyware distribution sites and keeps a cache to track cookies. The fifth false positive is `Microgarden`, a BHO that extends Internet Explorer with the ability to open multiple tabs in a single browser window. Although no suspicious API calls are invoked directly in response to events, this BHO makes use of timers. As a result, we have to conservatively consider all Windows API calls that this sample can possibly call (among which, a number of suspicious functions are found). The last three false positive examples suggest that the static analysis of Windows API calls may not deliver optimal detection results. Instead, one should seek to combine the results of our static analysis with those of our dynamic analysis to lower the number of false positives.

Taking a step back, a simple dynamic technique to identify spyware (Strategy 3) is to classify all BHO and toolbar components as malicious if they register as event sinks. As expected, all of the spyware samples receive browser events from Internet Explorer to monitor user behavior. In comparison, only eight of the 18 benign samples registered as event sinks. This observation suggests that many benign applications use BHO and toolbar extensions to improve Internet Explorer, but do not need to listen to events to implement their functionality. On the other hand, nearly half of the benign samples also use event information, for example, to display or modify the source of visited pages or to block pop-up windows.

For Strategy 4, the dynamic analysis is extended to monitor the interaction of the BHO or toolbar with the web browser. As mentioned previously, this is realized by recording the invocation of COM functions provided by the `IWebBrowser2` interface. To compile the list of suspicious COM functions, we analyzed this interface for functions that allow a browser extension to obtain information about the page or the location that a user is visiting. The complete list is shown in Figure 3. Of particular interest is the `get_Document()` method, which provides an `IHTMLDocument2` pointer to the Document Object Model (DOM) object of the web page that is being displayed. Using this pointer, a BHO or toolbar can modify a page or extract information from its source.

```
get_Document()
get_LocationURL()
get_LocationName()
```

Figure 3: *A priori* assembled list of suspicious COM browser functions.

Using the list of suspicious COM functions, dynamic analysis classifies a sample as spyware when at least one invocation of a suspicious function is observed in response to events. Unfortunately, this also results in more false positives than necessary. The reason is that several browser extensions interact with the browser in response to events. For example, the `Lost Goggles` toolbar requests a pointer to the DOM object of a loaded page to integrate thumbnails into search results returned by Google.

In our characterization of spyware, we claim that a malicious component both monitors user behavior and leaks this information to the environment. Thus, we expect the lowest number of false positives when employing a combination of dynamic and static analysis techniques. This is indeed confirmed by the detection results shown in Table 1 for Strategy 5. Compared to the results delivered by static analysis only, the misclassification of the benign `Spybot` and `T-Online` samples is avoided. The reason for this is that although these browser extensions might invoke a `WriteFile` API call in response to an event, the dynamic analysis confirms that they are not monitoring user behavior by calling any of the suspicious COM functions. `Microgarden` is also correctly classified as benign. Even though this toolbar uses timers, it does not access any relevant information in response to events. `Airoboform` and `P3P Client`, on the other hand, are still classified as spyware. The reason is that in addition to suspicious API calls, they also request the location of loaded pages via the `get_LocationURL()` function. However, as discussed previously, this is no surprise as these BHOs do indeed monitor surfing behavior and store (possibly sensitive) user information in files.

Table 2 shows the various execution times for each step in the analysis on a 1.7 GHz Pentium M processor with 1 GB of RAM. The execution time for dynamic analysis may be slower than one might expect. This is due to the fact that this analysis must be done in a virtual environment because we must execute the possibly malicious code. Furthermore, once this code is invoked the performance of the machine tends to degrade significantly. The execution time for static analysis, on the other hand, is split in two. This is because the running time for static analysis is highly dependent on how many events a sample is listening for. Thus, we give the execution time for disassembly and CFG creation along with a separate measure for the execution time to analyze a single event. We consider these performance measures to be acceptable for a prototype analysis tool and note that the running times could be significantly improved with optimization.

9.1 API Call Blacklist Derivation

Until now, we have been using lists of suspicious Windows API calls and suspicious COM functions that were generated *a priori*. An alternate method, as discussed in Section 4, is to generate these lists automatically. More precisely, by applying our approach to both a set of known benign samples and a set of known malicious samples, one can cross-reference the two resulting sets of calls made in response to browser events (from both static and dynamic analysis), to identify calls that are frequently observed for spyware, but never observed for benign BHOs or toolbars.

The major benefit of the automatic list generation approach is that it obviates the need to generate a list of suspicious calls *a priori*. Also, over time, as more samples are collected and analyzed, the list will become more refined, eliminating those calls that show up only in malicious samples by chance, and revealing new functions that were not considered before. These results are useful even in the case where one uses a list of calls generated *a priori* as the basis for detection, because there are a plethora of Windows API calls to consider, and the analysis can be used to update the "suspicious function" list with new calls as they begin to be utilized by spyware.

In the following, we briefly discuss our experience when automatically generating the list of suspicious Windows API functions. We refrained from applying automatic generation to the list of suspicious browser COM functions. The reason is that the `IWebBrowser2` COM interface contains considerably less functions than the Windows API and these functions are well-documented, making this list more suitable for *a priori* compilation. Figure 4 shows an excerpt of where the Windows API list we generated *a priori* and the list we generated automatically converged (a), as well as some additional malicious API calls that were discovered (b). These lists do indeed match up well with our initial intuition. Interestingly, new calls that we did not originally consider,

Analysis Step	Execution Time	
	Mean	Standard Deviation
1. Dynamic Analysis	30.97s	21.61s
2. Static Analysis (disassembly and CFG generation)	64.86s	137.94s
3. Static Analysis (per event CFG analysis)	80.01s	100.01s

Table 2: Performance for different analysis steps.

CreateProcessA
fclose
fopen
fwrite
Netbios
OleRun
InternetCanonicalizeA
InternetOpenUrlA
InternetSetCookieA
InternetSetOptionA
RegEnumKeyA
TerminateThread
WSACleanup
WSAStartup

(a)

FreeEnvironmentString
GetEnvironmentString
GetEnvironmentVariable
GetSystemDirectoryA
GetSystemInfo
CreateToolhelp32Snapshot
Process32First
Process32Next
SetupIterateCabinetA

(b)

Figure 4: Excerpts of extracted calls that (a) also appear in the *a priori* list and (b) are unique to the automatically derived list.

such as `CreateToolHelp32Snapshot`, which takes a snapshot of the processes currently running on a system and should probably not be called in response to browser events, can be added to the list of possibly malicious calls. The results indicate that our static list does a good job of detecting spyware, while our generated list can be used to further improve detection results as spyware authors try to adapt in order to evade detection.

Automated list generation, however, is not without its drawbacks. The reason is that we will likely be removing certain calls that *do* represent possible malicious intent. For example, when applied to our evaluation set, one of these calls would be the Windows API function `WriteFile`. Because `WriteFile` appears in both our benign and malicious sets of samples, we would disregard it as a common call that should not be taken into account when analyzing new and possibly malicious samples. This should reduce the number of false positives, but at the same time, it could result in an increase in the number of false negatives.

10 Conclusions and Future Work

Spyware is becoming a substantial threat to networks both in terms of resource consumption and user privacy violations. Current anti-spyware tools predominately use signature-based techniques, which can easily be evaded through obfuscation transformations.

In this paper, we present a novel characterization for a popular class of spyware, namely those components based on Browser Helper Objects (BHOs) or toolbars developed for Microsoft's Internet Explorer. This characterization is based on the observation that a spyware component first obtains sensitive information from the browser and then leaks the collected data to the outside environment. We developed a prototype detection tool based on our characterization that uses a composition of dynamic and static analysis to identify the browser COM functions and the Windows API calls that are invoked in response to browsing events. Based on this information, we are able to identify an entire class of spyware, thus making our approach more powerful than standard signature based techniques. In addition, our technique will provide a forensic analyst with detailed information about the behavior of unknown browser helper objects and toolbars.

Our approach was evaluated on a large test set of spyware and benign browser extensions. The results demonstrate that the approach is able to effectively identify the behavior of spyware programs without any *a priori* knowledge of the programs' binary structure, significantly raising the bar for malware authors who want to evade detection.

Future work will focus on extending our approach to spyware programs that do not rely on the Browser Helper Object or toolbar interfaces to monitor the user's behavior. We also plan to extend our characterization with more sophisticated data-flow analysis that would allow one to characterize the type of information accessed (and leaked) by the spyware program. This type of characterization would enable a tool to provide an assessment of the level of "maliciousness" of a spyware program.

Acknowledgments

This research was supported by the Austrian Science Foundation (FWF), under grant No. P18157, the Secure Business Austria competence center, the U.S. Army Research Office, under agreement DAAD19-01-1-0484, and by the National Science Foundation, under grants CCR-0238492 and CCR-0524853.

References

[1] A hidden menace. The Economist, June 2004.

[2] Ad-Aware. `http://www.lavasoftusa.com/software/adaware/`, 2005.

[3] Steven D. Gribble Alexander Moshchuk, Tanya Bragin and Henry M. Levy. A Crawler-Based Study of Spyware on the Web. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2006.

[4] M. Christodorescu and S. Jha. Testing Malware Detectors. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 34–44, Boston, MA, July 2004.

[5] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R.E. Bryant. Semantics-Aware Malware Detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland 2005)*, Oakland, CA, USA, May 2005.

[6] Data Rescure. IDA Pro: Disassembler and Debugger. `http://www.datarescue.com/idabase/`, 2005.

[7] Earthlink and Webroot Release Second SpyAudit Report. `http://www.earthlink.net/about/press/pr_spyAuditReport/`, June 2004.

[8] Aaron Hackworth. Spyware. US-CERT publication, 2005.

[9] Jan Hertsens and Wayne Porter. Anatomy of a Drive-By Install- Even on Firefox. `http://www.spywareguide.com/articles/anatomy_of_a_drive_by_install__72.%html`, 2006.

[10] Galen Hunt and Doug Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–144, Seattle, WA, 1999.

[11] C. Kruegel, W. Robertson, and G. Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 91–100, Tucson, AZ, December 2004.

[12] C. Kruegel, F. Valeur, W. Robertson, and G. Vigna. Static Analysis of Obfuscated Binaries. In *Proceedings of the Usenix Security Symposium*, 2004.

[13] C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *ACM Conference on Computer and Communications Security (CCS)*, 2003.

[14] Microsoft. Windows AntiSpyware (Beta): Analysis approach and categories. `http://www.microsoft.com/athome/security/spyware/software/isv/analysis.%mspx`, March 2005.

[15] Known Vulnerabilities in Mozilla Products. `http://www.mozilla.org/projects/security/known-vulnerabilities.html`, 2006.

[16] M. Oberhumer and L. Molnar. UPX: Ultimate Packer for eXecutables. `http://upx.sourceforge.net/`, 2004.

[17] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998.

[18] M. Pietrek. Peering Inside the PE: A Tour of the Win32 Portable Executable File Format. Microsoft Systems Journal, March 1994.

[19] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the USENIX LISA '99 Conference*, Seattle, WA, November 1999.

[20] S. Saroiu, S.D. Gribble, and H.M. Levy. Measurement and Analysis of Spyware in a University Environment. In *Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, March 2004.

[21] S. Schreiber. *Undocumented Windows 2000 Secrets: A Programmer's Cookbook*. Addison-Wesley Professional, 2001.

[22] Spybot Search & Destroy. `http://www.safer-networking.org/`, 2005.

[23] R. Thompson. Why Spyware Poses Multiple Threats to Security. *Communications of the ACM*, 48(8), August 2005.

[24] Y. Wang, R. Roussev, C. Verbowski, A. Johnson, M. Wu, Y. Huang, and S. Kuo. Gatekeeper: Monitoring Auto-Start Extensibility Points (ASEPs) for Spyware Management. In *Proceedings of the Large Installation System Administration Conference (LISA)*, Atlanta, GA, November 2004. USENIX.

[25] S. Willliams and C. Kindel. The Component Object Model: A Technical Overview. Microsoft Technical Report, October 1994.

[26] Onload XPI installs should be blocked by default. `https://bugzilla.mozilla.org/show_bug.cgi?id=238684`, 2004.

An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data

Nick L. Petroni, Jr.[†]
npetroni@cs.umd.edu
Department of Computer Science

Timothy Fraser[†]
tfraser@umiacs.umd.edu
Institute for Advanced Computer Studies

AAron Walters[‡]
arwalter@cs.purdue.edu
Department of Computer Science

William A. Arbaugh[†]
waa@cs.umd.edu
Department of Computer Science

† *University of Maryland, College Park, MD 20742, USA*
‡ *Purdue University, West Lafayette, IN 47907, USA*

Abstract

The ability of intruders to hide their presence in compromised systems has surpassed the ability of the current generation of integrity monitors to detect them. Once in control of a system, intruders modify the state of constantly-changing dynamic kernel data structures to hide their processes and elevate their privileges. Current monitoring tools are limited to detecting changes in nominally static kernel data and text and cannot distinguish a valid state change from tampering in these dynamic data structures. We introduce a novel general architecture for defining and monitoring *semantic integrity* constraints using a specification language-based approach. This approach will enable a new generation of integrity monitors to distinguish valid states from tampering.

1 Introduction

The foundation of the Trusted Computing Base (TCB) [26] on most currently deployed computer systems is an Operating System that is large, complex, and difficult to secure. Upon penetrating a system, sophisticated intruders often tamper with the Operating System's programs and data to hide their presence from legitimate administrators and to provide backdoors for easy re-entry. The Operating System kernel itself is a favored target, since a kernel modified to serve the attacker renders user-mode security programs ineffective. Many so-called ”rootkits” are now available to automate this tampering.

Recent advances in defensive technologies, such as external kernel integrity monitors [17, 37, 13, 29] and code attestation/execution verification architectures [18, 34, 33], have demonstrated their ability to detect the kinds of tampering historically performed by rootkits. Unfortunately, rootkit technology has already moved to a more sophisticated level. While these defensive technologies have focused on the relatively straightforward task of detecting tampering in static and unchanging regions of kernel text and data structures—typical targets of the previous generation of rootkits—the new rootkit generation has evolved to more sophisticated tampering behavior that targets dynamic parts of the kernel. Seeking to avoid detection and subsequent removal from the system, clever intruders can hide their processes from legitimate administrators by modifying links in the Linux and Windows XP/2000 kernels' process tables. Because the state of the process table changes continuously during kernel runtime, identifying these modified links is difficult for the current generation of kernel integrity monitoring tools that focus only on static data. Although this targeting of dynamic data was not entirely unanticipated by researchers [37, 13], there has yet to be a general approach for dealing with this threat.

In response to a continually advancing threat, we introduce an architecture for the runtime detection of *semantic integrity* violations in objects dynamically allocated in the kernel heap or in static objects that change depending upon the kernel state. This new approach is the first to address the issue of dynamic kernel data in a comprehensive way. In order to be effective against the latest rootkit technology, defensive mechanisms must consider both static and dynamic kernel data, as changes in either can lead to the compromise of the whole. We believe our approach provides an excellent complement to state of the art binary integrity systems.

Our approach is characterized by the following properties:

Specification-based. The previous generation's detection methods, which can be characterized by calculating hashes of static kernel data and text and comparing the result to known-good values, is not applicable to the continuously changing dynamic data structures now being targeted by rootkits. Instead of characterizing a correct state using hashes, our architecture relies upon an expert to describe the correct operation of the system via an abstract model for low-level data structures and the relationships between them. This model is a simplified description of security-relevant data structures and how they interoperate. Additionally, part of the specification is a set of constraints that must hold at runtime in order for the system to remain correct with regard to the *semantic integrity* of the kernel.

Automatic. The architecture includes a compiler that automatically translates the high-level specification language into low-level machine code to perform the checks. This automation allows experts to maximize the use of their time writing the specification and verifying its correctness, rather than writing low-level code.

Independent. Our architecture does not depend upon the correctness of the monitored kernel in order to detect that something is wrong. Instead, our approach relies on a trustworthy monitor that has direct access to kernel memory on the protected system and does not rely on the protected kernel's correctness.

Monitor agnostic. While our prototype implementation utilizes a PCI-based kernel monitor similar to Copilot [29] as the low-level mechanism for accessing system resources, our architecture allows for the use of any monitor with access to kernel memory that can also provide isolation. Other possibilities include software-based systems such as Pioneer [33] or a virtual machine introspection approach [13]. The focus of this work is on the *type* of checks performed, not the mechanism used to perform them. As such, our architecture is general enough to support different types of monitors, both software- and hardware-based.

Extensible response. The architecture is designed to allow specification writers to decide how the system should react to the violation of a particular constraint. At a minimum, most cases will require administrator notification. Currently, this is the only response we have implemented. However, the possibility for extension to other responses is apparent, particularly given the amount of forensic information available to our monitor.

We have demonstrated the feasibility of our approach by writing sample specifications for two different kernel subsystems in the Linux 2.6 kernel: the process (task) accounting system and the SELinux [22] mandatory access control (MAC) system's access vector cache (AVC). We have tested the system's effectiveness at detecting real-world attacks on dynamic kernel data in each subsystem, including a publicly available rootkit for the Linux kernel. Our results show that low-level code based on our initial specifications successfully detects the example attacks, which include data-only process hiding and modifications of SELinux access control results directly in memory.

2 Threats Against Dynamic Kernel Data

This section describes two examples of how intruders might, after gaining full administrative control of a GNU/Linux system, modify some of the kernel's dynamic data structures to their advantage. In the first example, an intruder removes tasks from the Linux kernel's all-tasks list in order to hide them from the system's legitimate administrators. In the second example, an intruder modifies an entry in the Linux kernel's SELinux access vector cache to temporarily elevate their privileges and disable auditing without making visible changes to the SELinux policy configuration. Note that neither of these examples expose flaws in the Linux kernel or its SELinux security module. These examples represent the potential acts of an intruder who has already gained full control of the system—perhaps by exploiting the trust or carelessness of the system's human operators in a manner entirely outside the scope of the system's technological safeguards.

2.1 Data-only Process Hiding

Rootkits have evolved beyond the historical methods of hiding processes, which included modifying the text of the `ps` program to lie to legitimate administrators or causing the kernel itself to lie by replacing the normally-static values of kernel text or function pointers, such as the system call vector or jump tables in the `/proc` filesystem, with the addresses of malicious functions. Even the most sophisticated threats became easy to detect by monitors that could compare the modified values against a known-good value—after all, in a healthy system, these values should never change [29].

Unfortunately, attackers do not need to modify any kernel code to hide processes within a running kernel. In fact, they do not need to rely on manipulating the control flow of the kernel at all. Instead, adversaries have found techniques to hide their processes even from correct, unmodified kernel code. By directly manipulating the underlying data structures used for process accounting, an attacker can quickly and effectively remove any desired process from the view of standard, unmodified administrator tools. While the process remains hidden for accounting purposes, it continues to execute as normal and will remain unaffected from the perspective of the scheduler. To understand how this state is achieved, we provide a brief overview of Linux 2.6 process management.

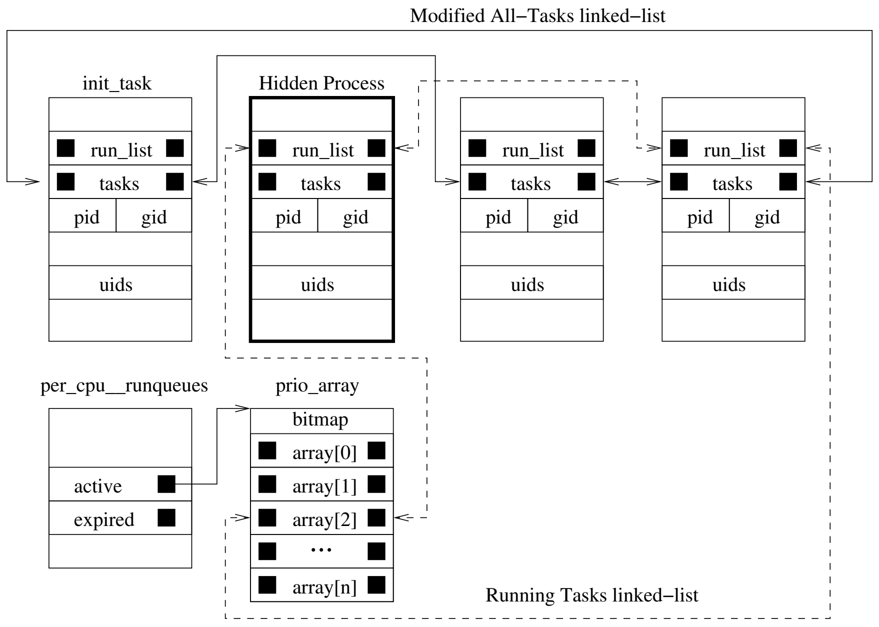


Figure 1: Data-only process hiding in Linux.

The primary data structure for process management in the Linux kernel is the `task_struct` structure [23]. All threads are represented by a `task_struct` instance within the kernel. A single-threaded process will therefore be represented internally by exactly one `task_struct`. Since scheduling occurs on a per-thread basis, a multi-threaded processes is simply a set of `task_struct` objects that share certain resources such as memory regions and open files, as well as a few other properties including a common process identifier (PID), the unique number given to each running process on the system.

In a correctly-running system, all `task_struct` objects are connected in a complex set of linked lists that represent various groupings relevant to that task at a particular time [23]. For accounting purposes, all tasks are members of a single doubly-linked list, identified by the `task_struct.tasks` member. This list, which we refer to as the all-tasks list, insures that any kernel function needing access to all tasks can easily traverse the list and be sure to encounter each task exactly once. The head of the task list is the `swapper` process (PID 0), identified by the static symbol `init_task`. In order to support efficient lookup based on PID, the kernel also maintains a hash table that is keyed by PID and whose members are hash-list nodes located in the `task_struct.pid` structure. Only one thread per matching hash of the PID is a member of the hash table; the rest are linked in a list as part of `task_struct.pid` member. Other list memberships include parent/child and sibling relationships and a set of scheduler-related lists discussed next.

Scheduling in the Linux kernel is also governed by a set of lists [23]. Each task exists in exactly one state. For example, a task may be actively running on the processor, waiting to be run on the processor, waiting for some other event to occur (such as I/O), or waiting to be cleaned up by a parent process. Depending on the state of a task, that task will be a member of at least one scheduling list somewhere in the kernel. At any given time, a typical active task will either be a member of one of the many wait queues spread throughout the kernel or a member of a per-processor run queue. Tasks cannot be on both a wait queue and a run queue at the same time.

Primed with this knowledge of the internals of Linux process management, we now describe the trivial technique by which an attacker can gain the ultimate stealth for a running process. Figure 1 depicts the primary step of the attack: removing the process from the doubly-linked all-tasks list (indicated by the solid line between tasks). Since this list is used for all process accounting functions, such as the `readdir()` call in the `/proc` filesystem, removal from this list provides all of the stealth needed by an adversary. For an attacker who has already gained access to kernel memory, making this modification is as simple as modifying two pointers per hidden process. As a secondary step to the attack, adversaries might also choose to remove their processes from the PID hash table (not pictured) in order to prevent the receipt of unwanted signals.

As shown in Figure 1, a task not present in the all-tasks list can continue to function because the set of lists used for scheduling is disjoint from the set used for accounting. The dashed line shows the relationship between objects relevant to a particular processor's run queue, including tasks that are waiting to be run (or are currently running) on that processor. Even though the second depicted task is no longer present in the all-tasks list, it continues to be scheduled by the kernel. Two simple changes to dynamic data therefore result in perfect stealth for the attacker, without any modifications to static data or kernel text.

2.2 Modification of System Capabilities

When most actions occur in the kernel, some form of a capability is used to identify whether or not a principal should be given (or already has been given) access to a resource. These capabilities therefore represent a prime target for attackers wishing to elevate privilege. Changing process user identifiers (UIDs) has long been a favorite technique of attackers. Other examples include file descriptors and sockets (both implemented in the same abstraction in the kernel).

The SELinux access vector cache provides a good example of this kind of capability and represents a potential target for an adversary seeking privilege escalation. This section describes the structure and purpose of the AVC and how an adversary might tamper with its state. Section 4 describes an experiment that demonstrates such tampering and the effectiveness of a prototype monitor for detecting this tampering.

SELinux [22] is a security module for Linux kernels that implements a combination of Type Enforcement [3] and Role-based [11] mandatory access control, now included in some popular GNU/Linux distributions. During runtime, SELinux is responsible for enforcing numerous rules governing the behavior of processes. For example, one rule might state that the DHCP [10] client daemon can only write to those system configuration files needed to configure the network and the Domain Name Service [24], but no others. By enforcing this rule, SELinux can limit the damage that a misbehaving DHCP client daemon might cause to the system's configuration files should it be compromised by an adversary (perhaps due to a buffer overflow or other flaw).

To enforce its rules, SELinux must make numerous decisions during runtime such as "Does the SELinux configuration permit this process to write this file?" or "Does it permit process A to execute program B?" Answering these questions involves some overhead, so SELinux includes a component called the access vector cache to save these answers. Whenever possible, SELinux rapidly retrieves answers from the AVC, resorting to the slower method of consulting the policy configuration only on AVC misses.

On our experimental system, the AVC is configured to begin evicting least frequently used entries after reaching a threshold of 512 entries. Our single-user system never loaded the AVC much beyond half of this threshold—although it was occasionally busy performing builds, these builds tended to pose the same small number of access control questions again and again. However, one could imagine a more complex multi-user system that might cause particular AVC entries to appear and disappear over time. Installations that permit SELinux configuration changes during runtime might also see AVC entries evicted due to revocation of privileges.

SELinux divides all resources on a system (such as processes and files) into distinct classes and gives each class a numeric Security Identifier or "SID." It expresses its mandatory access rules in terms of what processes with a particular SID may and may not do to resources with another SID. Consequently, at a somewhat simplified abstract level, AVC entries take the form of tuples:

```
<ssid, tsid, class, allowed, decided,
       audit-allow, audit-deny>
```

The `ssid` field is the SID of the process taking action, the `tsid` field is the SID of the resource the process wishes to act upon, and the `class` field indicates the kind of resource (file, socket, and so on). The `allowed` field is a bit vector indicating which actions (read, write, and so on) should be allowed and which should be denied. Only some of the `allowed` field bits may be valid—for example, if the questions answered by SELinux so far have involved only the lowest-order bit, then that may be the only bit that contains a meaningful 0 or 1. SELinux may or may not fill in the other `allowed` field bits until a question concerning those bits comes up. To distinguish a 0 bit indicating "deny" from a 0 bit indicating "invalid," the `decided` field contains a bit vector with 1 bits for all valid positions in the `allowed` field. The `audit-allow` and `audit-deny` fields are also bit vectors; they contain 1 bits for operations that should be logged to the system logger when allowed or denied, respectively.

It is conceivable that adversaries who have already gained administrative control over a system might wish to modify the SELinux configuration to give their processes elevated privileges. Certainly, they could accomplish this most directly by modifying the SELinux configuration files, but such modifications would be easily detected by filesystem integrity monitors like Tripwire [19]. Alternately, they might modify the in-kernel data structures representing the SELinux configuration—the same data structures SELinux consults to service an AVC miss. However, these data structures change in-frequently, when administrators decide to modify their SELinux configuration during runtime. Consequently, any tampering might be discovered by a traditional kernel integrity monitor that performs hashing or makes comparisons with correct, known-good values.

The state of the AVC, on the other hand, is dynamic and difficult to predict at system configuration time. Entries come and go with the changing behavior of processes. An adversary might insert a new AVC entry or modify an old one to effectively add a new rule to the SELinux configuration. Such an entry might add extra `allowed` and `decided` field bits to grant additional privileges, or remove existing `audit-allow` and `audit-deny` field bits to turn off troublesome logging. Such an entry would override the proper in-memory and on-disk SELinux configuration for as long as it remained in the cache. On a single-user installation like our experimental system, it would face little danger of eviction. On a busier system, frequent use might keep it cached for as long as needed.

3 The Specification Architecture

Our approach for detecting semantic integrity violations in dynamic kernel data structures is to define a high-level security *specification* [20] for kernel data that provides a simplified but accurate representation of how kernel objects in memory relate to one another, as well as a set of constraints that must hold on those data objects for the integrity of the kernel to remain intact. The result is a methodology that allows experts to concentrate on high-level concepts such as identifying security-relevant constraints, rather than writing low-level code to parse kernel data structures. The architecture we propose is composed of the following five components:

- *A low-level monitor*. The monitor is the entity that provides access to kernel memory at runtime. While there are a number of possible implementations, the primary requirement is consistent access to all of kernel virtual memory without reliance on the correctness of the protected kernel. Monitors that provide synchronous access to kernel memory, such as virtual machine monitors [13] or verifiable code execution [33], provide consistent views of kernel data, but run on the same host as the protected system and must contend with local applications for processor time. Asynchronous monitors typically have their own dedicated processor [29, 37, 17], but must make sense of snapshots of kernel memory that catch data structures in a temporarily-inconsistent mid-update state. In addition, monitors with access to system registers can protect themselves against attempts to bypass the monitor via malicious register changes [33].

- *A model builder*. The model builder is responsible for taking raw data from the low-level monitor and turning that data into the model abstraction defined by the specification, which is an input to the model builder. Effectively, the model builder is the bridge between the "bits" in kernel memory and the abstract objects defined by the user.

- *A constraint verifier*. As described above, the goal of the system is to apply high-level constraints to an abstract model of kernel data. The constraint verifier operates on objects provided by the model builder to determine if the constraints identified by the specification are met.

- *Response mechanisms*. When a constraint is violated, there is a security concern within the system. Depending on the nature of the violated constraint, an administrator may wish to take actions varying from logging an error to notifying an administrator or even shutting down the system. The constraint verifier is aware of the available response mechanisms and initiates those mechanisms according to the response determined by the specification.

- *A specification compiler*. Specifications are written in a high-level specification language (or languages) that describes the model, the constraints, and the responses to violated constraints. The specification compiler is responsible for turning the high-level language into a form that can be used by the model builder and the constraint verifier.

As shown in Figure 2, the first four of these are runtime components that work together to assess the integrity of a running kernel based on the input specification. The specification compiler is an offline component used only at the time of system setup or when specification updates are required. The primary logic of the monitor is driven by the constraint verifier, which iterates through all constraints to verify each in order. To facilitate the verification of each constraint, the verifier requests a consistent subset of the model from the model builder, which either has the information readily available or uses the low-level monitor to re-build that portion of the model. If a constraint passes, the verifier simply continues to the next. Failed constraints cause the verifier to dispatch a response mechanism according to the specification.

We now describe several aspects of the system in more detail, focusing primarily on the requirements for each component.

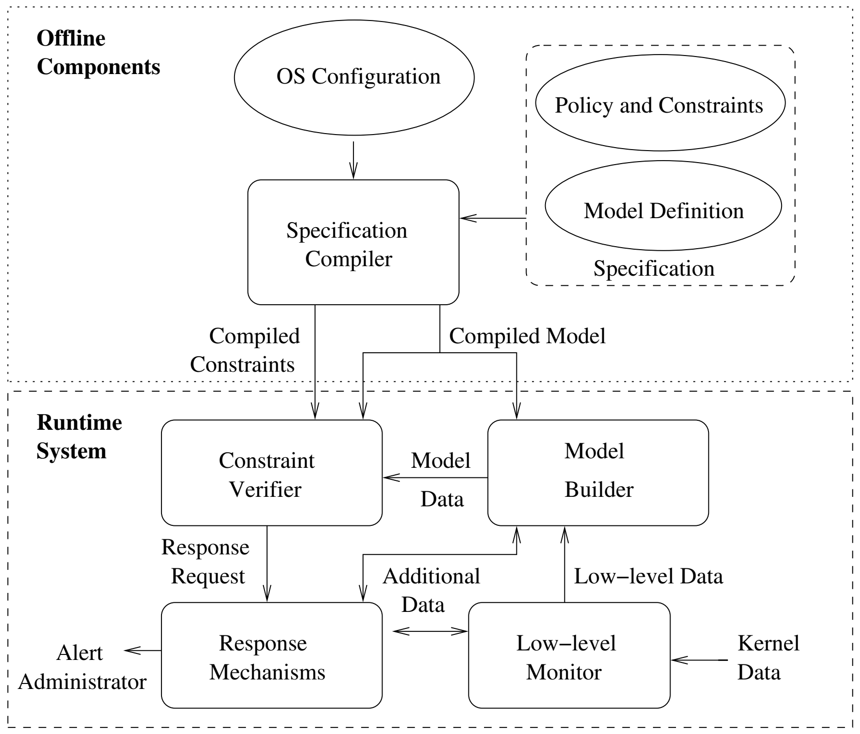


Figure 2: The semantic integrity monitor architecture.

3.1 Modeling Kernel Data

The concept of modeling low-level kernel data structures can be loosely thought of as a form of "inverted software design." A software designer's job is typically to take a high-level abstraction or set of real-world objects and represent those objects digitally in the system. One of the complex tasks for the programmer is efficiently and correctly representing real-world relationships among digital objects. Object modeling languages like the Unified Modeling Language (UML) [1] seek to aid the designer by providing formal constructs with which to define objects. In our system, the goal of the model specification writer is to abstract on the programmer's choice of data structures in order to describe the relevant relationships among digital objects. The resulting model allows an expert to write constraints at a high enough level of abstraction to express relevant object relationships without getting caught up in low-level details. To this end, the choice of modeling language directly affects the types of constraints that can and cannot be expressed on the model. Modeling languages that fail to capture relevant details of the underlying system will not allow potentially important constraints to be expressed. Similarly, modeling languages that provide too much expressive power on the underlying data will make the job of constraint writing overly complex. As a convenience, rather than inventing our own modeling language, we have chosen to reuse the data structure specification language created by Demsky and Rinard [7]. After redefining the language's semantics for our domain, the syntax proved effective for our example kernel data specifications with only minor modifications. We discuss these example kernel data specifications in Section 4.

It should be noted that model specifications correspond to a particular version (or versions) of the kernel. Therefore, as updates are made to kernel subsystems, so must the specification be updated. However, once a specification is written for a given kernel version, it can be shared and used at any deployed location. Furthermore, the specification compiler takes into account site-specific kernel configuration and symbol information to allow more widespread use of the specification. Finally, the relationships described in the specification will not change frequently and, even when they do change, will rarely change significantly enough to invalidate the entire specification. Tools for automating and improving the specification process are an area for future work.

3.2 Writing Model Constraints

At a high level, constraints are the underlying logic that determine whether or not the kernel is secure with regard to integrity. Constraints are therefore expressions of predicates reflecting invariant relationships among kernel objects represented in the model. Conceptually, constraints can be divided into two classes: those that are inherent to the correct operation of the system and those that represent site-specific policy. For example, the hidden process example described previously is clearly a violation of kernel integrity in any running kernel. However, one can envision a set of external constraints on kernel data objects that do not relate explicitly to the "correct" operation of the kernel as it was designed by kernel developers, but rather to conditions that an administrator has deemed should never occur on that machine. One example of such a constraint would be a requirement that no shell processes have user id zero (root). The requirements for a good constraint language include easy expression of properties of the underlying model, an ability to specify conditions under which a constraint must hold, and a mechanism for assigning a response to any violated constraint. To match our choice of initial modeling language, we have adapted Demsky and Rinard's constraint language to meet the needs described here [7].

Similar to the model specification, the constraints that must hold for a system may change when kernel developers make changes. However, like model specifications, constraints can be distributed for use at any deployment where a given model is valid.

3.3 Automating the System

One of the fundamental goals of our architecture is to relieve the engineering difficulties related to dynamic kernel data constraint checking and allow the expert to focus on security-relevant relationships among kernel objects. To this end, automating the model builder is a critical step. The primary responsibility of the specification compiler is to provide the model builder with a description of how digital objects should be turned into abstract objects and how those abstract objects are related. As in Demsky and Rinard's work [7], we propose that the specification compiler utilize automatic code generation to automate the model building and constraint checking processes. However, unlike the environment in which Demsky and Rinard's system functioned, the likely response for our system when a constraint fails is not repair. In fact, there may be reasons *not* to immediately fix the integrity violation so that more forensic information can be obtained without the attacker becoming aware that he or she has been detected. Furthermore, unlike Demsky and Rinard, in our system we do not have the benefit of executing within the running program that we are checking. Memory accesses are not free and pointer values are not local. In our system, every pointer dereference requires read operations by the low-level monitor. For these reasons, optimizing for repair is not the best approach for our environment. Rather, optimizing for efficient object accesses is more appropriate. Finally, performing checks asynchronously with the running kernel adds some additional challenges.

For a system that is externally analyzing a running kernel, the design of the model builder is non-trivial due to the complications of constantly changing data within the kernel. The assumptions that can be made by the model builder are closely tied to the properties of the low-level monitor. However, assuming a monitor that is running asynchronously relative to the protected kernel, the following are a minimal set of design considerations for the model builder and specification compiler components:

- How will the system distinguish inconsistent data resulting from a read that occurs while the kernel is in the middle of a data structure update from an invalid kernel state?
- How can the system schedule data reads such that relationships to be tested among digital objects are tested on a set of objects that were read at or about the same time?
- How can the system schedule data reads to minimize the total number of reads necessary to check a particular constraint or set of constraints?

In Section 4, we discuss how our initial implementation handles these issues. To summarize our results, we postulate that simple extensions to the modeling language can help the specification compiler reason about the nature of underlying data, including how likely it is to change over time and the best order in which to process it. As a promising indication of our success, the resulting system experienced no false positives in any of our tests. However, in a specification-based system the possibility for false positives or false negatives is more a reflection of the specification than of the system. An expert with better knowledge of the system will have more success in this regard.

4 Implementation

In this section, we describe our implementation of the above architecture and the testing we performed on a system running the Fedora Core 4 GNU/Linux distribution. Using our system, we have implemented (in C) two specifications designed to protect the Linux 2.6 process accounting and SELinux AVC subsystems respectively. We then tested our specifications against implementations of the two attacks described in Section 2. We successfully detected both of these attacks with zero false positives when our detection code was running on a PCI-based monitor similar to Copilot [29]. Table 1 provides more detailed information about our test environment. These

	Protected Host	PCI-based Monitor
Machine Type	Dell Dimension 4700	Bus-mastering PCI add-in card
RAM	1GB	32MB
Processor	Single 2.8GHz Pentium 4	200MHz Motorola PowerPC 405GP
Storage	40GB IDE Hard Disk	4MB Flash memory
Operating System	Redhat Fedora Core 4 full installation	Embedded Linux 2.4 kernel
Networking	10/100 PCI NIC	10/100 on-board NIC

Table 1: Semantic integrity test platform summary.

tests demonstrate that it is possible to write useful specifications using our technique, and that these specifications can be coupled with an existing integrity monitor to provide an effective defense against real attacks.

We begin our discussion by describing our specification language, an adaptation of that presented by Demsky and Rinard [7], in the context of our Linux process accounting example.

4.1 Writing Specifications: a Linux Hidden Process Example

Demsky and Rinard introduced a system for automatically repairing data structure errors based on model and constraint specifications [7]. The goal of their system was to produce optimized data structure error detection and repair algorithms [9] that were guaranteed to terminate [8]. Because of the differences explained in Section 3, we have adapted Demsky and Rinard's specification languages and the corresponding parser and discarded all of the automatic code generation portions. Our intention is to replace them with a code generation algorithm better suited to our environment. This section provides a brief overview of their specification language syntax and identifies the changes necessary to support our kernel integrity system. It also introduces our first example specification for detecting hidden processes in the Linux kernel. Demsky and Rinard's specification system is actually composed of four separate languages:

Low-level Structure Definition: The structure definition language provides C-like constructs for describing the layout of objects in memory. Demsky and Rinard provide a few additions to the normal C language syntax. First, fields may be marked "reserved," indicating that they exist but are not used. Second, array lengths may be variable and determined at runtime through expression evaluation. Third, a form of structure "inheritance" is provided for notational simplicity whereby structures can be defined based on other structures and then expanded with additional fields. We found no need to change the structure definition language syntax developed by Demsky and Rinard. However, it was necessary to adapt the language's semantics in two important ways because of the "external" nature of our monitor.

First, named structure instances, which are also declared in the structure definition language, cannot be resolved because our monitor is not part of the normal software linking process. Instead, we must use an external source for locating variables. Our current implementation allows the user to provide these locations manually or to have them extracted automatically from a Linux `System.map` symbol table file. The second semantic modification necessary for the structure definition language is the handling of pointer values, which are not "local" to our monitor. Instead, pointers must be treated as foreign addresses accessed through the monitor's memory access mechanism.

Figure 3(a) contains our specification of the Linux kernel's process accounting data structures written in the structure definition language. Figure 3(b) contains the result of a manual translation from this specification into the corresponding C declarations that will become part of the monitoring code. Note the use of the `host_addr_t` to represent host addresses after byte-order conversion on the monitor. As described above, the appropriate value for the `LINUX_SYMBOL_init_task` constant (and other required symbols) is automatically extracted from the Linux `System.map` symbol table file by our configuration tool.

Model Space Definition: The second language, shown in Figure 3(c) for our process accounting example, defines a group of sets or relations (there are no relations in our first example) that exist in the model [7]. There are two sets in our specification: one corresponding to all processes in the all-tasks list (the `AllTasks` set) and one corresponding to all processes in the run queue (the `RunningTasks` set). Both are of type `Task` in the model. We made no modifications to this simple language, as all of our example specifications were able to be expressed in the context of sets and relations. The model space definition language provided by Demsky and Rinard also provides support for set partitions and subsets.

```
Task init_task;

structure Task {
      reserved byte[32];
      ListHead run_list;
      reserved byte[52];
      ListHead tasks;
      reserved byte[52];
      int pid;
      reserved byte[200];
      int uid;
      reserved byte[60];
      byte comm[16];
}

structure ListHead {
      ListHead *next;
      ListHead *prev;
}

sructure Runqueue {
      reserved byte[52];
      Task *curr;
}
```

(a) Low−Level Structure Definiton

```
host_addr_t init_task =
    LINUX_SYMBOL_init_task;
struct Task {
      unsigned char reserved_1[32];
      ListHead run_list;
      unsigned char reserved_2[52];
      ListHead tasks;
      unsigned char reserved_3[52];
      int pid;
      unsigned char reserved_4[200];
      int uid;
      unsigned char reserved_5[60];
      unsigned char comm[16];
};

struct ListHead {
      host_addr_t next;
      host_addr_t prev;
};

struct Runqueue {
      unsigned char reserved_1[52];
      host_addr_t curr;
};
```

(b) Translated Structure Definiton

```
set AllTasks(Task);
set RunningTasks(Task);
```

(c) Model Space Definition

```
[ for_circular_list i as ListHead.next starting init_task.tasks.next ], true => container(i, Task,tasks.next) in AllTasks;
[ ], true => runqueue.curr in RunningTasks;
```

(d) Model Building Rules

```
[ for t in RunningTasks ],  t  in  AllTasks
      : notify_admin("Hidden task " + t.comm + " with PID " + t.pid + " detected at kernel virtual address " + t);
```

(e) Constraints

Figure 3: Process accounting subsystem specification.

Model Building Rules: Thus far we have discussed languages for describing the low-level format and organization of data in kernel memory and for declaring the types of high-level entities we will use in our model. The model building rules bridge the gap between these by identifying which low-level objects should be used within the abstract model. These rules take the form

```
    [<quantifiers>], <guard> ->
         <inclusion rule>;
```

For each rule, there is a set of quantifiers that enumerates the objects to be processed by the rule, a guard that is evaluated for each object to determine if it should be subject to the rule, and an inclusion that determines how that object should be classified in the abstract model. We have made the following (syntactic and semantic) modifications to Demsky and Rinard's model building language:

1. *User-defined rule order.* In Demsky and Rinard's system, the specification compiler could identify the dependencies among rules and execute them in the most appropriate order. Furthermore, their denotational semantics required execution of the rule function until a least fixed point was reached. This approach is not suited for external monitors for two reasons. First, because memory accesses are of a much higher performance penalty in our system, the expert benefits from the ability to describe which objects should be read in which order to build a complete model. Second, unlike in Demsky and Rinard's environment, the low-level monitor may be performing its reads asynchronously with the monitored system's execution. Model building accesses that have not been optimized are more likely to encounter inconsistent data as the system state changes.

2. *Pointer handling.* As previously mentioned, pointer references are not local in our environment and must go through the low-level monitor's memory access system. To detect invalid pointers, Demsky and Rinard developed a runtime system that instruments the heap allocation and deallocation (`malloc()`, `free()`, etc.) functions to keep

track of valid memory regions. This approach is clearly not an option for external monitors, which are not integrated with the system's runtime environment. Currently, invalid pointers are handled by restarting the model build process. If the same invalid pointer is encountered during two consecutive model build operations, an error is generated. If the invalid pointer is not encountered again, it is assumed the first error was an inconsistency stemming from the asynchronous nature of the monitor.

3. *The* `contains()` *expression.* A common programming paradigm (especially in the Linux kernel) is to embed generic list pointer structures as members within another data structure. Our added expression gives specification writers an easy way to identify the object of which a particular field is a member.

4. *The* `for_list` *quantification.* Linked lists are a common programming paradigm. This expression gives specification writers a straightforward way to indicate they intend to traverse a list up to the provided stop address (or NULL if not indicated).

5. *The* `for_circular_list` *quantification.* This is syntactic sugar for the `for_list` construct where the end address is set equal to the first object's address. The Linux kernel makes heavy use of circular lists.

Figure 3(d) shows the model rules for our process accounting example. The first rule indicates that a circular list starting (and ending) at `init_task.tasks.next` will be processed. The keyword `true` in the guard indicates that all members of this list should be subject to the inclusion. The inclusion itself uses our `container()` expression to locate the `Task` that contains the list pointer and to include that `Task` in AllTasks. The second rule is very simple; it creates a singleton set `RunningTasks` with the current task running on the run queue.

Constraints: The final part of the specification defines the set of constraints under which the model is to be evaluated. The basic form of a rule in Demsky and Rinard's constraint language is as follows [7]:

```
[ <quantifiers> ], <predicate>;
```

In the constraint language, the set of quantifiers may include only sets defined in the model. The predicate is evaluated on each quantified member and may include set operations and evaluations of any relations defined in the model. If the predicate fails for any quantified member, Demsky and Rinard's system would seek to repair the model (and the underlying data structures accordingly). In our system, however, we have added a "response" clause to the end of the constraint rule as follows:

```
[ <quantifiers> ], <predicate> :
    <[consistency,] response>;
```

This critical extension allows the specification writer to dictate how failures are to be handled for a particular rule. In addition to identifying which action to take, the response portion allows for an optional "consistency parameter." This parameter allows the specification writer to identify a "safe" number of failures before taking action and helps prevent false positives that might occur due to data inconsistencies. If no such parameter is provided, the default value of two consecutive failures is used. Of course, a secondary result is that actual rule violations will be given an opportunity to occur once without detection. The specification writer will need to balance the advantages and the disadvantages for each constraint rule and can always disable this feature by setting the value to zero. For the threat considered in our Linux process accounting example, the default value is acceptable because of the nature of the targeted threat. A process that is short-lived has no reason to hide, since an administrator is unlikely to notice the process. Finally, the consistency value has no meaning for synchronous monitors, which do not suffer from the same consistency problems.

Figure 3(e) shows the single constraint rule for our hidden process example. The rule states that if any process is ever seen running on the processor that is not in the all-tasks list, we have a security problem and need to alert the administrator. This example describes a relatively simple method of detecting hidden processes. In order to detect a hidden process, the monitor must catch the process while it has the host CPU—a probabilistic strategy that is likely to require the taking of many snapshots of the host's state over time before the hidden process's luck runs out. A more deterministic approach might be to compare the population of the kernel's numerous wait and run queues with the population of the all-tasks list. In order to be eligible for scheduling, a process must be on one of these wait or run queues; a process on a wait or run queue but not in the all-tasks list is hiding. This strategy would require a more complex model specification.

4.2 A Second Example: the SELinux AVC

In Section 2, we described an attack against the SELinux AVC whereby an attacker with the ability to write to memory could modify the permissions of an entry in

```
 SidTab sidtab;
 AVCCache avc_cache;
 Policydb policydb;

structure ListHead {
      ListHead *next;
      ListHead *prev;
}
structure AVCCache {
      ListHead slots[512];
}

structure Policydb {
      reserved byte[108];
      AVTab te_avtab;
      rserved byte[8];
      AVTab te_cond_avtab;
}
```

```
structure AVCNode {
      int ssid;
      int tsid;
      short tclass;
      reserved short;
      int allowed;
      int decided;
      int auditallow;
      int auditdeny;
      int seqno;
      int atomic;
      ListHead list;
}
```

```
structure AVTab {
      AVTabNode **htable;
      int nel;
}

structure AVTabNode {
      int source_type;
      int target_type;
      int target_class;
      int specified;
      int allowed;
      int auditdeny;
      int auditallow;
      AVTabNode *next;
}
```

```
structure SidTab {
      SidTabNode **htable;
      int nel;
}

structure SidTabNode {
      int sid;
      int user;
      int role;
      int type;
      reserved byte[24];
      SidTabNode *next;
}
```

(a) Low−Level Structure Definitons

```
Set AllSids(SidTabNode);                      avcssidtype : AVCNode −> SidTabNode;
Set AllAVCNodes(AVCNode);                     avctsidtype : AVCNode −> SidTabNode;
Set TEAVTabNodes(AVTabNode);                  avcteavtabmapping : AVCNode −> TEAVTabNode;
Set TECondAVTabNodes(AVTabNode);              avctecondavtabmapping : AVCNode −> TECondAVTabNode;
```

(b) Model Space Definition

Figure 4: SELinux access vector cache structure and model definitions.

the cache to give a particular process access not permitted by the SELinux policy. We further explained that existing hashing techniques can be used to protect the memory-resident full policy, but not the AVC because of its dynamic nature. Our approach for protecting the AVC therefore begins with the assumption that a simple "binary" integrity system is protecting the static data structures that represent the full policy. We then use our semantic integrity monitor to implement a specification whose goal is to compare all AVC entries with their protected entries in the full policy. Figures 4 and 5 display the full specification we used to protect the SELinux AVC. This specification is more complex than the previous example largely due to the complexities of the SELinux system and its data structures. However, the complexity of the specification is minimal as compared with the number of lines of code that would be required to implement the equivalent checks in low-level code (eight model definition rules and one constraint rule versus the 709 lines of C code in our example implementation).

There are four primary entities in our SELinux specification: the security identifier table (of type SIDTab), the access vector cache (an AVCCache), the Type Enforcement access vector table (an AVTab), and its counterpart the Type Enforcement conditional access vector table (also an AVTab). The model definition rules first create a set of SIDs by walking through the SID table and then, similarly, create a set of all AVC nodes from the AVC. The third and fourth rules are used to create mappings between the AVC nodes and their source and target SIDs. Rules five and six look-up each AVC node in the full Type Enforcement policy for both conditional and non-conditional access vector tables. The final two model definition rules create a mapping between AVC nodes and their corresponding entries in the Type Enforcement access vector tables. The single constraint rule simply walks through all AVC nodes and checks that the allowable field matches the combined (bitwise OR) value of the two corresponding Type Enforcement access vector entries for that AVC node. As with the last example, an administrator is notified if the data structures are found to be inconsistent.

We have tested our code against an attacking loadable kernel module that modifies the permissions for a particular AVC entry. A rootkit might make such a modification to temporarily elevate the privileges of one or more processes in a manner that could not be detected by an integrity monitor that observed only static data structures. Our specification successfully detects the attack against our Fedora Core 4 system configured with the default SELinux "targeted" policy operating in "enforcing" mode.

```
[ for i = 0 to 128, for_list j as SidTabNode.next starting sidtab.htable[i] ], true => j in AllSids ;
[ for i = 0 to 512, for_circular_list j as ListHead.next starting avc_cache.slots[i] ], true =>
      true => container (j, AVCNode, list.next ) in AllAVCNodes ;
[ for a in AllAVCNodes, for s in AllSids ], (a.ssid = s.sid) => <a,s> in avcssidtype ;
[ for a in AllAVCNodes, for s in AllSids ], (a.tsid  = s.sid) => <a,s> in avctsidtype ;
[ for a in AllAVCNodes, for_list j as AVTabNode.next starting
      policydb.te_avtab.htable[a.tclass + a.avctsidtype.type * 4 + a.avcssidtype.type * 512].next ],
      (j.source_type = a.avcssidtype.type AND j.target_type = a.avctsidtype.type)
            => j in TEAVTabNodes;
[ for a in AllAVCNodes, for_list j as AVTabNode.next starting
      policydb.te_cond_avtab.htable[a.tclass + a.avctsidtype.type * 4 + a.avcssidtype.type * 512].next ],
      (j.source_type = a.avcssidtype.type AND j.target_type = a.avctsidtype.type)
            => j in TECondAVTabNodes;
[ for c in AllACNodes, for a in TEAVTabNodes ],
      (c.avcssidtype.sid = a.source_type AND
       c.avctsidtype.sid = a.target_type AND
       c.tclass = a.target_class) =>
         <c,a> in avcteavtabmapping;
[ for c in AllACNodes, for a in TECondAVTabNodes ],
      (c.avcssidtype.sid = a.source_type AND
       c.avctsidtype.sid = a.target_type AND
       c.tclass = a.target_class) =>
         <c,a> in avctecondavtabmapping;
```

(a) Model Building Rules

```
[ for c in AllAVCNodes ], c.allowed = (c.avcteavtabmapping.allowed | c.avctecondavtabmapping.allwed)
      : notify_admin ("AVC Cache entry has improper privileges " + c.callowed + " at virtual address " + c);
```

(b) Constraints

Figure 5: SELinux access vector cache specification rules.

5 Discussion

The approach proposed in this paper is to detect malicious modifications of kernel memory by comparing actual observed kernel state with a specification of correct kernel state. The specification describes possible correct kernel states, not signatures of known attacks. In this way, our approach is a type of specification-based intrusion detection. We do not follow the approach of traditional signature-based virus scanners. Thus far, we have provided two example specifications for our system and identified the types of modifications that these specifications can detect. While our examples are useful for demonstrating how the proposed system works, they provide little intuition about how specifications would be developed in a real deployment. In this section, we provide a high-level methodology for identifying system properties of interest and describe three classes of threats we have identified.

Currently, there are two methods for identifying data properties and writing their corresponding specifications: (1) analyzing and abstracting on known threats and (2) deriving data properties and specifications from a high-level English-language security policy. In the analysis of known threats, the goal is to classify the techniques used by adversaries in previous attacks in order to abstract on these methodologies. The result is the identification of a set of data invariants that may be violated by future attacks. Of course, this approach permits the possibility that new attacks may avoid detection by exploiting only those details of the kernel abstracted out of the specification, leading to an interminable ”arms race” between attackers and specification-writers. Nevertheless, this approach is still better than the traditional signature-based virus-scanning approach in that each specification has the potential to detect an entire class of similar attacks, rather than only a single instance.

It may be possible to avoid such an arms race by using an alternate approach: deriving specifications from a high-level English-language security policy rather than from an analysis of known attacks. In this approach, an analyst might begin with a policy such as ”no runnable processes shall be hidden” or ”my reference monitor enforces my particular mandatory access control policy” and then examine the kernel source to determine which data structures have relevant properties and what those properties should be in order for the high-level policy to hold. The analyst's task is similar to constructing a formal argument for correctness, except that the end result is a configuration for a runtime monitor.

Section 4 presents two examples of the types of specifications one might obtain as a result of the methodologies just described. Using these techniques, we have identified three classes of attacks against dynamic kernel data. While it is likely there are other classes of attacks, we believe the three identified thus far provide evidence of the potential success of our approach. The following are the attack classes we have identified:

Data hiding attacks. This class of attacks was demonstrated in Section 2.1 with the Linux process hiding example. The distinguishing characteristic of this class is the removal of objects from data structures used by important kernel subsystems for accounting and reporting. Writing specifications capable of detecting these attacks requires identifying data structures that are used by kernel resource reporting procedures such as system calls and, in the case of Linux, the `/proc` filesystem.

Capability/access control modification attacks. One of the fundamental goals of kernel attackers is to provide their processes with privileges and access to resources. To this end, process capabilities in the form of tokens, flags, and descriptors are likely targets of an attacker with kernel memory access. In addition to the SELinux AVC example, described in Section 2.2, we have identified user/group identifiers, scheduler parameters (e.g., nice value), and POSIX capabilities as potential targets. We are actively writing specifications to protect this data.

Control flow modification attacks. One popular technique for gaining control of kernel functionality is the modification of function pointers in dynamic data structures such as those associated with the virtual filesystem (VFS) and `/proc` filesystem. As demonstrated by popular rootkits like `adore-ng`, manipulating these pointers provides attackers with a "hook" to execute their inserted code. While previous generations of kernel integrity monitors have demonstrated effective detection of hooks placed in static data (e.g., the system call table), dynamic function pointers have remained an elusive target. We are actively writing a large number of simple specification rules to test the validity of kernel pointers throughout dynamic data. Additionally, we intend to investigate the use of automated tools to make this process easier and more complete.

Unlike misuse detection systems, our specification-based approach allows for the identification and detection of *classes* of attacks without a priori knowledge of particular instances of threats.

6 Related Work

The architecture we have proposed was inspired by the work of four separate areas: external kernel monitors [17, 37, 13, 29], specification-based intrusion detection [20, 32], specification-based data structure repair [7], and semantic integrity in database systems [15]. Work in the areas of software attestation and verifiable code generation is also closely related. We briefly describe this body of work here.

6.1 Kernel Integrity Monitors

We broadly categorize external kernel monitors as any system that operates outside of the protected kernel in order to provide independent, trustworthy analysis of the state of the protected host. Examples of such systems in the recent literature include coprocessor-based monitors [37, 29], SMP-based monitors [17], and virtual machine introspection [13]. While each of these systems has introduced its own mechanism for inspecting the internal state of the protected host, all have at least one common goal: to monitor a running host for unauthorized modifications of kernel memory. While some ad-hoc techniques for limited protection of dynamic data have been demonstrated (although not described in detail) on a couple of these systems [13, 37], the predominant detection technique remains binary or checksum comparison of known static objects in memory.

The types of checks performed by these systems are not incorrect or without value. These systems provide a foundation on which our approach aims to extend to broaden the set of kernel attacks detectable from such platforms.

6.2 Attestation and Verifiable Execution

Code attestation [18, 12, 34, 30, 31, 35] is a technique by which a remote party, the "challenger" or "verifier," can verify the authenticity of code running on a particular machine, the "attestor." Attestation is typically achieved via a set of measurements performed on the attestor that are subsequently sent to the challenger, who identifies the validity of the measurements as well as the state of the system indicated by those measurements [30]. Both hardware-based [12, 30, 31] and software-based [18, 34] attestation systems have been developed. Measurement typically occurs just before a particular piece of code is loaded, such as between two stages of the boot process, before a kernel loads an new kernel module, or when a kernel loads a program to be executed in userspace [30]. All of the hardware-based systems referenced in this paper utilize the Trusted Computing Group's (TCG) [2] Trusted Platform Module (TPM), or a device with similar properties, as a hardware root of trust that validates measurements prior to software being loaded at runtime. Software-based attestation systems attempt to provide similar guarantees to those that utilize trusted hardware and typically rely on well-engineered verification functions that, when modified by an attacker, will necessarily produce incorrect output or take noticeably longer to run. This deviation of output or running time is designed to be significant enough to alert the verifier of foul play.

Traditional attestation systems verify only binary properties of static code and data. In such systems, the only runtime benefit provided is the detection of illegal modifications that utilize well-documented transitions or interfaces where a measurement has explicitly been inserted before the malicious software was loaded. Unfortunately, attackers are frequently not limited to using only these interfaces [33].

Haldar et al. have proposed a system known as "semantic remote attestation" [14] in an attempt to extend the types of information the verifying party can learn about the attesting system. Their approach is to use a language-based trusted virtual machine that allows the measurement agent to perform detailed analysis of the application rather than simple binary checksums. The basic principle is that language-based analysis can provide much more semantic information about the properties of an application. Their approach does not extend to semantic properties of the kernel and, since their VM runs on top of a standard kernel, there is a requirement for traditional attestation to bootstrap the system.

Verifiable code execution is a stronger property than attestation whereby a verifier can guarantee that a particular piece of code actually runs on a target platform [33]. This contrasts traditional attestation, where only the loading of a particular piece of software can be guaranteed. Once that software is loaded however, it could theoretically be compromised by an advanced adversary. With verifiable code execution, such a modification should not be possible without detection by the verifier. Both hardware-based [5, 35] and, more recently, software-based [33] systems have been proposed.

Verifiable code execution is a promising direction for ensuring that the correct code is run on a potentially untrusted platform. As shown by Seshadri et al. [33], such a system could be used as the foundation for a kernel integrity monitor. We therefore view verifiable code execution as a potential monitor extension for our architecture.

6.3 Specification-Based Detection

Specification-based intrusion detection is a technique whereby the system policy is based on a specification that describes the correct operation of the monitored entity [20]. This approach contrasts signature-based approaches that look for known threats and statistical approaches for modeling normalcy in an operational system. Typically, specification-based intrusion detection has been used to describe program *behavior* [20, 21, 32] rather than correct state, as we have used it [20, 21, 32]. More recently, specifications have been used for network-based intrusion detection as well [36].

6.4 Data Structure Detection and Repair

We have already described Demsky and Rinard's [7] work towards data structure error detection and repair. This work places one level of abstraction on top of the historical 5ESS [16] and MVS [25] work: in those systems, the inconsistency detection and repair procedures were coded manually. We have utilized the basic techniques of Demsky and Rinard's specification system with the necessary adaptations for operating system semantic integrity. The environments are sufficiently different so as to require significant modifications. These differences were discussed in Section 3.3.

In similar work, Nentwich and others [27] have developed xlinkit, a tool that detects inconsistencies between distributed versions of collaboratively-developed documents structured in XML [4]. It does so based on consistency constraints written manually in a specification language based on first order logic and XPath [6] expressions. These constraints deal with XML tags and values, such as "every item in this container should have a unique name value." In later work [28], they describe a tool which analyzes these constraints and generates a set of repair actions. Actions for the above example might include deleting or renaming items with non-unique names. Human intervention is required to prune repair actions from the list and to pick the most appropriate action from the list at repair time.

6.5 Semantic Integrity in Databases

There is a long history of concern for the correct and consistent representation of data within databases. Hammer and McLeod addressed the issue in the mid 1970's as it applies to data stored in a relational database [15]. The concept of insuring transactional consistency on modifications to a database is analogous to that of doing process accounting within the operating system. The database, like the operating system, assumes that data will be modified only by authorized parties through pre-defined interfaces. While the environments are very different, Hammer and McLeod's work provided excellent insight to us regarding constraint verification. Their system includes a set of constraints over database relations that include an assertion (a predicate like in our system), a validity requirement (analogous to the guard in Demsky and Rinard's model language), and a violation action, similar to our response mechanism but which only applies to updating the database. Hammer and McLeod argue that assertions should not be general purpose predicates (like first-order logic), but should instead be well-defined.

7 Future Work

Each part of the architecture described above provides avenues for significant impact and advancement in the system. The three most promising areas are the extension to other monitors, advancement in system responses, and the analysis and automation of specifications.

We have designed the semantic integrity architecture to be easily extended to other monitor platforms. Two of the most promising such platforms include virtual machine monitors [13, 12] and software-based monitors achieved via verifiable code execution [33]. These systems provide the possibility for unique extensions such as the inclusion of register state into specifications and the benefit of added assurance without the need for extra hardware. It is our intention to extend our work to at least one such software-based monitor.

A second avenue of work we intend to pursue is that of additional response vectors. Having an independent monitor with access to system memory and a system for easily interpreting that memory can provide a huge amount of leverage for advanced response. Perhaps the most significant potential for work is the advancement of automated runtime memory forensics.

Finally, as with all security systems, having a good policy is very important for the success of the system. Our current architecture requires experts with advanced knowledge of kernel internals to write and verify specifications. Developing tools to help automate the process, including a number of kernel static analysis tools, could significantly improve this process. We intend to investigate techniques for analyzing kernel properties automatically, both statically and at runtime.

8 Conclusion

We have introduced a novel and general architecture for defining and monitoring *semantic integrity* constraints—functionality required to defeat the latest generation of kernel-tampering rootkit technology. For our initial prototype implementation, we have adapted Demsky and Rinard's specification languages for implementing internal monitors for application data structures [7] to the task of implementing external monitors for operating system kernel data structures. This adaptation required adding features to their specification languages to overcome a number of issues not present in the original application problem domain, including managing memory transfer overhead and providing for flexible responses to detected compromises.

Our general architecture is applicable to a variety of low-level monitoring technologies, including external hardware monitors [29], software-based monitors [33] and virtual machine introspection [13]. We believe our approach is the first to address the issue of monitoring the integrity of dynamic kernel data in a comprehensive way, and we believe it will provide an excellent complement to present state of the art binary integrity systems.

Acknowledgments

We would like to thank Trent Jaeger for his time and feedback during the final preparation of this work. We would also like to thank the anonymous reviewers for their helpful comments. This work was supported by the National Science Foundation (NSF) under CAREER award 0133092.

References

[1] The Unified Modeling Language (UML). `http://www.uml.org`, 2005.

[2] Trusted Computing Group (TCG). `http://www.trustedcomputinggroup.org`, 2005.

[3] W. E. Boebert and R. Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th National Computer Security Conference*, pages 18–27, Gaithersburg, Maryland, September 1985.

[4] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language. Recommendation REC-xml-20001006, World Wide Web Consortium, October 2000.

[5] B. Chen and R. Morris. Certifying Program Execution with Secure Processors. In *9th Workshop on Hot Topics in Operating Systems (HotOS)*, Lihue, Hawaii, May 2003.

[6] J. Clark and S. Derose. XML Path Language (XPath) Version 1.0. Recommendation REC-xpath-19991116, World Wide Web Consortium, November 1999.

[7] B. Demsky and M. Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Anaheim, CA, October 2003.

[8] B. Demsky and M. Rinard. Static Specification Analysis for Termination of Specification-Based Data Structure Repair. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, November 2003.

[9] B. Demsky and M. Rinard. Data Structure Repair Using Goal-Directed Reasoning. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, St. Louis, MO, May 2005.

[10] R. Droms. Dynamic host configuration protocol. Technical Report RFC 2131, Bucknell University, March 1997.

[11] D. Ferraiolo and R. Kuhn. Role-Based Access Controls. In *Proceedings of the 15th National Computer Security Conference*, pages 554–563, Baltimore, Maryland, October 1992.

[12] T. Garfink el, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual-Machine Based Platform for Trusted Computing. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, Sagamore, NY, October 2003.

[13] T. Garfink el and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *The 10th Annual Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, February 2003.

[14] V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation – a virtual machine directed approach to trusted computing. In *Proceedings of the 3rd USENIX Virtual Machine Research & Technology Symposium*, May 2004.

[15] M. Hammer and D. McLeod. A Framework For Data Base Semantic Integrity. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE)*, San Francisco, CA, October 1976.

[16] G. Haugk, F. Lax, R. Royer, and J. Williams. The 5ESS(TM) switching system: Maintenance capabilities. *AT & T Technical Journal*, 64 part 2(6):1385 – 1416, July-August 1985.

[17] D. Hollingworth and T. Redmond. Enhancing operating system resistance to information warfare. In *MILCOM 2000. 21st Century Military Communications Conference Proceedings*, volume 2, pages 1037–1041, Los Angeles, CA, USA, October 2000.

[18] R. Kennell and L. H. Jamieson. Establishing the Genuinity of Remote Computer Systems. In *Proceedings of the 12th USENIX Security Symposium*, pages 295–310, Washington, D.C., August 2003.

[19] G. H. Kim and E. H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 18–29, Fairfax, Virgina, November 1994.

[20] C. Ko, G. Fink, and K. Levitt. Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference (ACSAC)*, Orlando, FL, 1994.

[21] C. Ko, M. Ruschitzka, and K. Levitt. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach . In *1997 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997.

[22] P. A. Loscocco and S. D. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, Boston, Massachusetts, June 2001.

[23] R. Love. *Linux Kernel Development*. Novell Press, second edition, 2005.

[24] P. Mockapetris. Domain names—conceptsand facilities. Technical Report RFC 1034, ISI, November 1987.

[25] S. Mourad and D. Andrews. On the Reliability of the IBM MVS/XA Operating System. *IEEE Transactions on Software Engineering*, 13(10):1135–1139, 1987.

[26] National Computer Security Center. *Department of Defense Trusted Computer System Evaluation Criteria*, December 1985.

[27] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151 – 185, May 2002.

[28] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *Proceedings fo the 25th International Conference on Software Engineering*, May 2003.

[29] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In *13th USENIX Security Symposium*, San Diego, CA, August 2004.

[30] R. Sailer, T. Jaeger, X. Zhang, and L. van Doorn. Attestation-based Policy Enforcement for Remote Access. In *11th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, November 2004.

[31] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *13th USENIX Security Symposium*, San Diego, CA, August 2004.

[32] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *8th USENIX Security Symposium*, pages 63–78, Washington, D.C., August 1999.

[33] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, United Kingdom, October 2005.

[34] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: SoftWare-based ATTestation for Embedded Devices. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2004.

[35] E. Shi, A. Perrig, and L. V. Doorn. BIND: A Fine-grained Attestation Service for Secure Distributed Systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2005.

[36] C. Tseng, P. Balasubramanyam, C. Ko, R. Limprasittiporn, J. Rowe, and K. Levitt. A specification-based instrusion detection system for aodv. In *2003 ACM Workshop on security of Ad Hoc and Sensor Networks (SASN '03)*, Fairfax, VA, October 2003.

[37] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer. Secure Coprocessor-based Intrusion Detection. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

vTPM: Virtualizing the Trusted Platform Module

Stefan Berger   Ramón Cáceres   Kenneth A. Goldman
Ronald Perez   Reiner Sailer   Leendert van Doorn
{stefanb, caceres, kgoldman, ronpz, sailer, leendert}@us.ibm.com
IBM T. J. Watson Research Center
Hawthorne, NY 10532 USA

Abstract

We present the design and implementation of a system that enables trusted computing for an unlimited number of virtual machines on a single hardware platform. To this end, we virtualized the Trusted Platform Module (TPM). As a result, the TPM's secure storage and cryptographic functions are available to operating systems and applications running in virtual machines. Our new facility supports higher-level services for establishing trust in virtualized environments, for example remote attestation of software integrity.

We implemented the full TPM specification in software and added functions to create and destroy virtual TPM instances. We integrated our software TPM into a hypervisor environment to make TPM functions available to virtual machines. Our virtual TPM supports suspend and resume operations, as well as migration of a virtual TPM instance with its respective virtual machine across platforms. We present four designs for certificate chains to link the virtual TPM to a hardware TPM, with security vs. efficiency trade-offs based on threat models. Finally, we demonstrate a working system by layering an existing integrity measurement application on top of our virtual TPM facility.

1 Introduction

Hardware virtualization has enjoyed a rapid resurgence in recent years as a way to reduce the total cost of ownership of computer systems [5]. This resurgence is specially apparent in corporate data centers such as web hosting centers, where sharing each hardware platform among multiple software workloads leads to improved utilization and reduced operating expenses.

However, along with these cost benefits come added security concerns. Workloads that share the same platform must often be kept separate for a multitude of reasons. For example, government regulations may require an investment bank to maintain a strict separation between its market analysis and security underwriting departments, including their respective information processing facilities. Similarly, commercial interests may dictate that the web sites of competing businesses not have access to each other's data. In addition, concerns about malicious software subverting normal operations become specially acute in these shared hardware environments. For example, a remote client of a medical services site would like to determine that the server is not running corrupted software that will expose private information to a third party or return wrong medical information. The increasing use of virtualization thus gives rise to stringent security requirements in the areas of software integrity and workload isolation.

The combination of a hardware-based root of trust such as the Trusted Platform Module (TPM) [23], and a virtual machine-based system such as Xen [4], VMware [26], or PHYP [14], is exceedingly well suited to satisfying these security requirements. Virtual machine monitors, or hypervisors, are naturally good at isolating workloads from each other because they mediate all access to physical resources by virtual machines. A hardware root of trust is resistant to software attacks and provides a basis for reasoning about the integrity of all software running on a platform, from the hypervisor itself to all operating systems and applications running inside virtual machines.

In particular, the TPM enables remote attestation by digitally signing cryptographic hashes of software components. In this context, attestation means to affirm that some software or hardware is genuine or correct. TPM chips are widely deployed on laptop and desktop PCs, and are becoming increasingly available on server-class machines such as the IBM eServer x366 [12].

Virtualizing the TPM is necessary to make its capabilities available to all virtual machines running on a platform. Each virtual machine with need of TPM functionality should be made to feel that it has access to its own private TPM, even though there may be many more virtual machines than physical TPMs on the system (typically there is a single hardware TPM per platform). It is thus necessary to create multiple virtual TPM instances, each of which faithfully emulates the functions of a hardware TPM.

However, virtualizing the TPM presents difficult challenges because of the need to preserve its security properties. The difficulty lies not in providing the low-level TPM command set, but in properly supporting higher-level security concepts such as trust establishment. In particular, it is necessary to extend the chain of trust from the physical TPM to each virtual TPM via careful management of signing keys and certificates. As a result, some application and operating system software that relies on TPM functionality needs to be made aware of semantic differences between virtual and physical TPMs, so that certificate chains can be correctly built and evaluated, and trust chains correctly established and followed.

An additional challenge is the need to support migration of a virtual TPM instance between hardware platforms when its associated virtual machine migrates. The ability to suspend, migrate, and resume virtual machines is an important benefit of hardware virtualization. For the virtual TPM, migration requires protecting the secrecy and integrity of data stored in a virtual TPM instance during the transfer between platforms, and re-establishing the chain of trust on the new platform.

This paper presents the design and implementation of a virtual TPM (vTPM) facility. This work makes the following contributions:

- It identifies the requirements for a vTPM, including those related to migration between hardware platforms.
- It introduces a vTPM architecture that meets these requirements, including extensions to the standard TPM command set and a protocol for secure vTPM migration.
- It describes our implementation of this vTPM architecture on Xen, including support for remote integrity attestation of the complete system: boot loader, hypervisor, vTPM subsystem, operating systems, and applications.
- It discusses four alternative schemes for certifying a vTPM's security credentials, including the tradeoffs involved in choosing between them.
- It demonstrates that our vTPM facility works by running an existing TPM application inside Xen virtual machines.

This work can also serve as a template for how to virtualize other security-related devices such as secure co-processors. Virtualizing such devices presents similar challenges to those outlined above for TPMs.

The rest of this paper is organized as follows. Section 2 introduces background concepts useful for understanding the ensuing material. Section 3 presents the requirements on a virtual TPM facility. Sections 4, 5, and 6 respectively describe the design, the implementation, and a sample application of our vTPM facility. Section 7 discusses open issues, Section 8 covers related work, and Section 9 concludes the paper.

2 Background

In this section we give some background on the two technologies that are basic to understanding this paper: the Trusted Platform Module (TPM) and the Virtual Machine Monitor (VMM).

2.1 The Trusted Platform Module

The TPM is a security specification defined by the Trusted Computing Group [23]. Its implementation is available as a chip that is physically attached to a platform's motherboard and controlled by software running on the system using well-defined commands [11]. It provides cryptographic operations such as asymmetric key generation, decryption, encryption, signing and migration of keys between TPMs, as well as random number generation and hashing. It also provides secure storage for small amounts of information such as cryptographic keys. Because the TPM is implemented in hardware and presents a carefully designed interface, it is resistant to software attacks [3].

Of particular interest is the Platform Configuration Register (PCR) *extension* operation. PCRs are initialized at power up and can only be modified by reset or extension. The PCR extension function cryptographically updates a PCR using the following function:

$$Extend(PCR_N, value) = SHA1(PCR_N||value)$$

The cryptographic properties of the extension operation state that it is infeasible to reach a certain PCR state through two different sequences of values. SHA1 refers to the Secure Hash Algorithm standard [19]. The || operation represents a concatenation of two byte arrays.

PCR extensions are used during the platform boot process and start within early-executed code in the Basic Input/Output System (BIOS) that is referred to as the Core Root of Trust for Measurement (CRTM) [24]. Hash values of byte arrays representing code or configuration data are calculated, or *measured*, and PCRs are extended with these values. A final PCR value represents this accumulation of a unique sequence of measurements. Along with a sequential list of individual measurements and applications' names and information about measured configuration data, PCR values are used to decide whether a system can be trusted. A transitive trust model is implemented that hands off the measuring from the BIOS [24] to the boot loader [18] and finally to the operating system. Procedures have also been developed for operating systems to measure launched applications, scripts and configuration files [21].

Besides the aforementioned cryptographic operations it is possible to *seal* information against the state of the TPM, where its state is represented through a subset of PCRs. Sealed information is encrypted with a public key and can only be decrypted if the selected PCRs are in the exact state that they were at the time of sealing.

There are a number of signing keys associated with a TPM. Each TPM can be identified by a unique built-in key, the Endorsement Key (EK), which stands for the validity of the TPM [10]. The device manufacturer should provide a certificate for the EK. Related to the EK are Attestation Identity Keys (AIKs). An AIK is created by the TPM and linked to the local platform through a certificate for that AIK. This certificate is created and signed by a certificate authority (CA). In particular, a *privacy CA* allows a platform to present different AIKs to different remote parties, so that it is impossible for these parties to determine that the AIKs are coming from the same platform. AIKs are primarily used during quote operations to provide a signature over a subset of PCRs as well as a 160-bit nonce. Quotes are delivered to remote parties to enable them to verify properties of the platform.

2.2 Virtual Machine Monitors

VMMs [8], also known as hypervisors, allow multiple operating systems to simultaneously run on one machine. A VMM is a software layer underneath the operating system that meets two basic requirements:

- It provides a Virtual Machine (VM) abstraction that models and emulates a physical machine.
- It provides isolation between virtual machines.

The basic responsibility of a VMM is to provide CPU time, memory and interrupts to each VM. It needs to set up the page tables and memory management unit of the CPU such that each VM runs in its own isolated sandbox. The hypervisor itself remains in full control over the resources given to a VM. During the boot process of a VMM, often an initial virtual machine is started that serves as a management system for starting further virtual machines.

Depending on the fidelity of the emulation of a physical machine, it may be necessary to make modifications to an operating system for it to run on a VMM. If modifications are required the environment is said to be *paravirtualized*, otherwise the VMM is said to provide a *fully virtualized* environment.

3 Requirements

A virtual TPM should provide TPM services to each virtual machine running on top of a hypervisor. The requirements discussed in this section can be summarized as follows:

1. A virtual TPM must provide the same usage model and TPM command set to an operating system running inside a virtual machine as a hardware TPM provides to an operating system running directly on a hardware platform.
2. A strong association between a virtual machine and its virtual TPM must be maintained across the life cycle of virtual machines. This includes migration of virtual machines together with their associated virtual TPMs from one physical machine to another.
3. A strong association between the virtual TPM and its underlying trusted computing base (TCB) must be maintained.
4. A virtual TPM must be clearly distinguishable from a hardware TPM because of the different security properties of the two types of TPM.

As much software as possible that was originally written to interact with a hardware TPM should run unmodified with a virtual TPM. It should remain unaware of the fact that it is communicating with a software implementation of a TPM in a virtual environment. An example of software that should remain unmodified is the TCG Software Stack (TSS) [25] that issues low-level TPM requests and receives low-level TPM responses on behalf of higher-level applications.

The requirement that software be unaware that it is using emulated devices is basic to virtualization and has already been achieved for a wide range of devices found in modern computers. Open-source software such as QEmu [1], as well as proprietary products like VMWare Workstation [26], have been successful in emulating machine environments for personal computers. They provide transparent emulation for timers, interrupt controllers, the PCI bus, and devices on that bus.

However, as a security device the TPM presents new and challenging issues that preclude fully transparent virtualization. One challenge arises because modern virtual machine monitors provide suspend and resume capabilities. This enables a user to freeze the state of an operating system and resume it at a later point, possibly on a different physical machine. A virtual TPM implementation must support the suspension and resumption of TPM state, including its migration to another system platform. During normal operation of the virtual TPM, as well as during and after these more sophisticated lifecycle operations, the association between the virtual TPM and its virtual machine must be securely maintained such that secrets held inside the virtual TPM cannot be accessed by unauthorized parties or other virtual machines.

Another challenge is to maintain the association of a virtual TPM to its underlying trusted computing base. PC manufacturers may issue a certificate for the TPM endorsement key (EK) that states that the TPM hardware is tightly coupled to the motherboard and correctly embedded into the BIOS for management. A challenger, validating a digital signature from such a TPM, can thus determine the correct embedding and operation of the remote TPM chip and establish the environmental security properties of the hardware TPM. In a virtualized environment, each operating system communicates with a virtual TPM that may be running as a user-space process inside its own virtual machine. The association of such a TPM with its underlying software and hardware platform is not only loose but also subject to change, e.g., during migration. Tracking this changing trusted computing base forms one major challenge in virtualizing a hardware TPM. Maintaining the ability of the virtual TPM to attest to its mutable trusted computing base forms another major challenge. It is necessary to enable remote parties that have established trust in the initial environment to also establish trust in the vTPM environment at a later point in time.

For example, the strong binding of TPM credentials to those of the hardware platform is important to challenging parties during remote attestation. The challenger must follow the trust chain from the target platform's hardware TPM through a virtual TPM and into the runtime environment of the associated virtual machine.

Further, since software TPM implementations do not usually offer the same security properties as hardware TPM implementations, the different types of TPMs should be distinguishable for remote parties relying on a TPM's correct functioning. A virtualized TPM's certificates can be used to give an interested party enough information to conclude relevant properties of the complete software, firmware, and hardware environment on which this TPM's correct operation depends. In practice, this can be realized by the certificate issuer embedding special attributes into the certificate, and the interested party validating the certificate and translating these attributes during remote attestation of security properties. Interestingly, as will become clear during our exposition, a software TPM can be as secure as a hardware TPM .

In summary, virtualizing the TPM is not achieved by merely providing TPM functionality to a virtual machine through device emulation. A virtual TPM must also provide the means for outside parties to establish trust in a larger software environment than is the case with hardware TPMs. It must also enable reestablishment of trust after a virtual machine is migrated to another platform. These requirements for providing virtual TPM functionality will be used as a guideline for the following sections on architecture and implementation, as well as our final discussion.

4 Architecture

We designed a virtual TPM facility in software that provides TPM functionality to virtual machines. This section first describes the structure of the vTPM and the overall system design. It proceeds with describing our extensions to the TPM 1.2 command set to support virtualization of the TPM. Then it introduces our protocol for virtual TPM migration and concludes with considering security aspects of the vTPM platforms and run-time environments involved in the migration.

Figure 1 illustrates the vTPM building blocks and their relationship. The overall vTPM facility is composed of a vTPM manager and a number of vTPM instances. Each vTPM instance implements the full TCG TPM 1.2 specification [11]. Each virtual machine that needs TPM functionality is assigned its own vTPM instance. The vTPM manager performs functions such as creating vTPM instances and multiplexing requests from virtual machines to their associated vTPM instances.

Virtual machines communicate with the vTPM using a split device-driver model where a client-side driver runs inside each virtual machine that wants to access a virtual TPM instance. The server-side driver runs in the virtual machine hosting the vTPM.

4.1 Associating vTPM Instances with their Virtual Machines

As shown in Figure 1, multiple virtual machines send TPM commands to the virtual TPM facility. A difficulty arises because it cannot be determined from the content of a TPM command from which virtual machine the command originated, and thereby to which virtual TPM instance the command should be delivered. Our solution is for the server-side driver to prepend a 4-byte vTPM instance identifier to each packet carrying a TPM command. This number identifies the vTPM instance to which a virtual machine can send commands. The in-

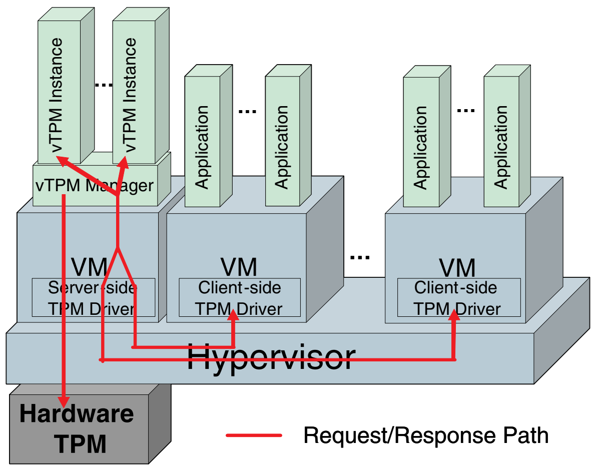


Figure 1: vTPM Architecture

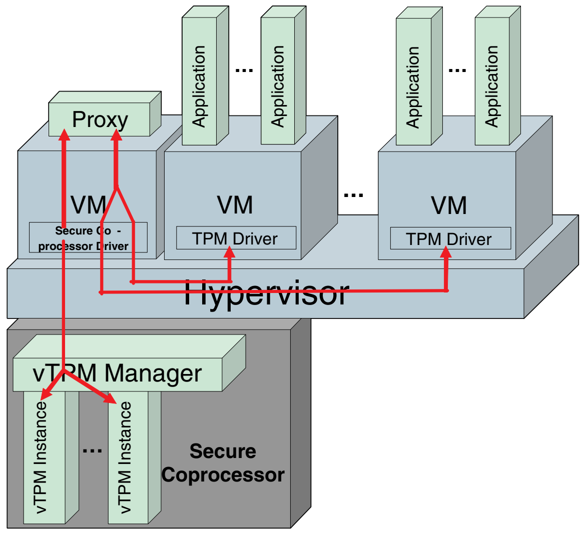


Figure 2: vTPM running inside a Secure Co-processor.

stance number is assigned by the virtual TPM when the vTPM instance is created.

Every VM must associate with a unique vTPM instance. The vTPM instance number is prepended on the server side so that virtual machines cannot forge packets and try to get access to a vTPM instance that is not associated with them. A command's originating virtual machine can be determined from the unique interrupt number raised by each client-side driver.

Since a TPM holds unique persistent state with secret information such as keys, it is necessary that a virtual machine be associated with its virtual TPM instance throughout the lifetime of the virtual machine. To keep this association over time, we maintain a list of virtual-machine-to-virtual-TPM-instance associations.

Figure 1 shows our architecture where TPM functionality for all VMs is provided by a virtual TPM running in the management VM. TPM functionality for this VM is provided by the hardware TPM, and is used in the same way as in a system without a hypervisor where the operating system owns the hardware TPM.

A variation of this architecture is shown in Figure 2 where virtual TPM functionality is provided by an external secure coprocessor card that provides maximum security for sensitive data, such as private keys, through a tamper-responsive environment. Here the first VM is the owner of this hardware and uses one virtual TPM instance for its own purposes. All other instances are reserved for usage by other virtual machines. A proxy process forwards TPM messages between the server-side driver and the external card.

4.2 The Root vTPM Instance

Our design was driven by the goal of having a virtual TPM implementation that can be run on an external coprocessor card as well as executed as a process running within a virtual machine. We designed the virtual TPM such that the interaction of applications with either implementation would be the same. New commands and APIs that we introduce should work the same for both implementations. Considerations regarding reducing the trusted computing base of the environment hosting the virtual TPM did not directly influence the design, although the intention is to have a virtual machine that is dedicated exclusively to providing virtual TPM functionality.

Further, modern hypervisors support advanced features such as virtual machine hibernation and migration from one physical platform to another. The straightforward approach to supporting such features is to hibernate and migrate a virtual TPM instance along with its associated virtual machine, thus preserving existing measurements and avoiding the complexity of remeasuring running software in a new environment and accounting for the loss of measurements representing software that was loaded but is no longer running. However, the virtual TPM migration process must offer more security guarantees for the virtual TPM instance state than is usually provided for an operating system image that is being transferred. The virtual TPM migration process must guarantee that any vTPM instance state in transit is not subject to modification, duplication, or other compromise.

This set of requirements led us to design a virtual TPM as a TPM capable of spawning new vTPM child instances. Having an always available vTPM *root in-*

stance provides an entity that has cryptographic capabilities for generating asymmetric keys, handling encryption and decryption of data, and migration of asymmetric keys between virtual TPMs. The ability to handle keys and encrypt data with them enables us to encrypt the state of a vTPM instance when migrating it. The virtual TPM's ability to migrate keys to another virtual TPM makes it possible to exchange encrypted data between virtual TPMs.

Since the ability to spawn – and generally to manage – new virtual TPM instances is a fairly powerful feature, this capability should only be accessible to the owner of the root instance. The administrator of the initial virtual machine, who has the ability to start new virtual machines, would own this capability. We designed all TPM command extensions to require owner authorization in the same way as some of the existing TPM commands do. In effect, the TPM verifies that such command blocks are authorized with the owner's password.

We introduced the concept of a privileged vTPM instance. A privileged vTPM instance can spawn and manage child vTPM instances. Since being a privileged instance is an inheritable property, an instance may pass this privilege on to its own children. Using this inheritance scheme, we can support building a hierarchy of vTPMs in parallel to a hierarchy of virtual machines where each virtual machine is given the privilege of starting other virtual machines.

4.3 Independent Key Hierarchies

The TPM specification demands that a TPM establish a storage root key (SRK) as the root key for its key hierarchy. Every key that is generated has its private key encrypted by its parent key and thus creates a chain to the SRK. In our virtual TPM we create an *independent key hierarchy* per vTPM instance and therefore unlink every vTPM instance from the key hierarchy of a possible hardware TPM. This has the advantage that key generation is much faster since we do not need to rely on the hardware TPM for this. It also simplifies vTPM instance migration.

Similarly, we generate an endorsement key (EK) per vTPM instance. This enables TPM commands that rely on decrypting information with the private part of the EK to also work after a virtual TPM has migrated.

If the SRK, EK or any other persistent data of virtual TPMs are written into persistent memory, they are encrypted with a symmetric key rooted in the hardware TPM by for example sealing it to the state of the hardware TPM's PCRs during machine boot, or by encrypting it using a password-protected key. We therefore earn the flexibility of managing each virtual TPM's key hierarchy independently. In addition, by using file-level data encryption we mitigate the cost of not directly coupling the key hierarchy of a virtual TPM instance to that of the hardware TPM.

4.4 Extended Command Set

In order to realize our design of a virtual TPM, we extended the existing TPM 1.2 command set with additional commands in the following categories.

- Virtual TPM Management commands

  CreateInstance

  DeleteInstance

  SetupInstance

- Virtual TPM Migration commands

  GetInstanceKey / SetInstanceKey

  GetInstanceData / SetInstanceData

- Virtual TPM Utility commands

  TransportInstance

  LockInstance / UnlockInstance

  ReportEnvironment

The *Virtual TPM Management commands* manage the life-cycle of vTPM instances and provide functions for their creation and deletion. The *SetupInstance* command prepares a vTPM instance for immediate usage by the corresponding virtual machine and extends PCRs with measurements of the operating system kernel image and other files involved in the boot process. This command is used for virtual machines that boot without the support of a TPM-enabled BIOS and boot loader, which would otherwise initialize the TPM and extend the TPM PCRs with appropriate measurements.

The *Virtual TPM Migration commands* support vTPM instance migration. We implemented a secure virtual TPM instance migration protocol that can securely package the state of a virtual TPM instance and migrate it to a destination platform. Our extended commands enforce that the content of a vTPM instance is protected and that a vTPM instance can only be migrated to one target platform destination, thus preventing duplication of a vTPM instance and ensuring that a virtual TPM is resumed in association with its VM.

One of the *Virtual TPM Utility commands* offers a function for routing a limited subset of TPM commands from a vTPM parent instance to one of its child instances. This command works similar to IP tunneling, where an embedded packet is unwrapped and then routed to its destination. Embedding a command is useful since the association of a virtual machine to a privileged virtual TPM does not allow direct communication with a child vTPM instance. For example, we use this command to create an endorsement key for a virtual TPM after the child instance has been created and before it is used by its associated virtual machine. Other functions in the utility category include locking a vTPM instance to keep its state from being altered while its state is serialized for migration, and unlocking it to make it available for use after migration has completed.

4.5 Virtual TPM Migration

Since vTPM instance migration is one of the most important features that we enabled through the command set extension, we explain how it works in more detail. The virtual TPM migration procedure is depicted in Figure 3.

We enabled vTPM instance migration using asymmetric and symmetric keys to encrypt and package TPM state on the source virtual TPM and decrypt the state on the destination virtual TPM. We based vTPM migration on migrateable TPM storage keys, a procedure that is supported by the existing TPM standard.

The first step in our vTPM instance migration protocol is to create an empty destination vTPM instance for the purpose of migrating state. The destination virtual TPM generates and exports a unique identifier (Nonce). The source vTPM is locked to the same Nonce. All TPM state is exported with the Nonce, and the Nonce is validated before import. This enforces uniqueness of the virtual TPM and prevents TPM state from being migrated to multiple destinations.

The next step involves marshaling the encrypted state of the source vTPM. This step is initiated by sending to the source vTPM a command to create a symmetric key. The key is encrypted with a parent TPM instance storage key. The blobs of state encrypted with a symmetric key are then retrieved from the source vTPM. This includes NVRAM areas, keys, authorization and transport sessions, delegation rows, counters, owner evict keys, and permanent flags and data. While the state is collected, the TPM instance is locked so the state cannot be changed by normal usage. After each piece of state information has been serialized, an internal migration digest is updated with the data's hash and the piece of state information becomes inaccessible. The migration digest is embedded into the last piece of state information and serves for validation on the target side.

To recreate the state of the virtual TPM on the destination platform, the storage key of the vTPM parent instance (used to encrypt the symmetric key used to protect the vTPM instance state) must be migrated to the destination vTPM parent instance. After the decryption of the symmetric key, the migrating vTPM's state is recreated and the migration digest recalculated. To detect possible Denial of Service (DoS) attacks where untrusted software involved in migration alters or omits state, operation of the vTPM instance can only resume if the calculated migration digest matches the transmitted one.

Support for Live Migration Modern virtual machine monitors support *live migration* [2] of virtual machines from one platform to another. Live migration tries to shorten downtimes by replicating the running system's image on a destination machine and switching execution to that machine once all pages have been replicated. Live migration can be supported with our virtual TPM migration protocol, but will in the worst case extend the downtime of the migrated system by the time it takes to complete an outstanding TPM operation, transfer the vTPM state, and recreate it on the destination platform.

4.6 Linking a vTPM to its TCB

Both architectures we introduced in Section 4.1 – a vTPM hosted in a virtual machine or in a secure coprocessor – provide TPM functionality to virtual machines. It is therefore possible to enable an integrity measurement facility [13] in each virtual machine and record application measurements in the virtual TPM. However, it is necessary that a challenger can establish trust in an environment which consists of more than the content of the virtual machine. The reason is that each operating system is running inside a virtual machine that is fully controlled by the hypervisor. Furthermore, a virtual TPM can be running as a process inside a VM whose own execution environment must be trusted. Therefore it is necessary that attestation support within the virtualized environment not only allows a challenger to learn about measurements inside the virtual machine, but also about those of the environment that provides virtual TPM functionality. In addition, these measurements must include the hypervisor and the entire boot process.

Our architecture therefore merges the virtual TPM-hosting environment with that of the virtual machine by providing two different views of PCR registers. Figure 4 shows these two views. The lower set of PCR registers of a vTPM show the values of the hardware TPM and the upper ones reflect the values specific to that vTPM. This way, a challenger can see all relevant measurements. The providers of the measurements extended into the different PCRs –BIOS, boot loader, and operating system– are denoted beside the PCRs. BIOS measurements include measurements of the boot stages and various hardware platform configurations. The boot loader measures, for example, the hypervisor and its configuration, the virtual machine monitor operating system kernel, initrd, and configuration. Then the VMM takes over and measures the dynamically activated VMM environment, such as

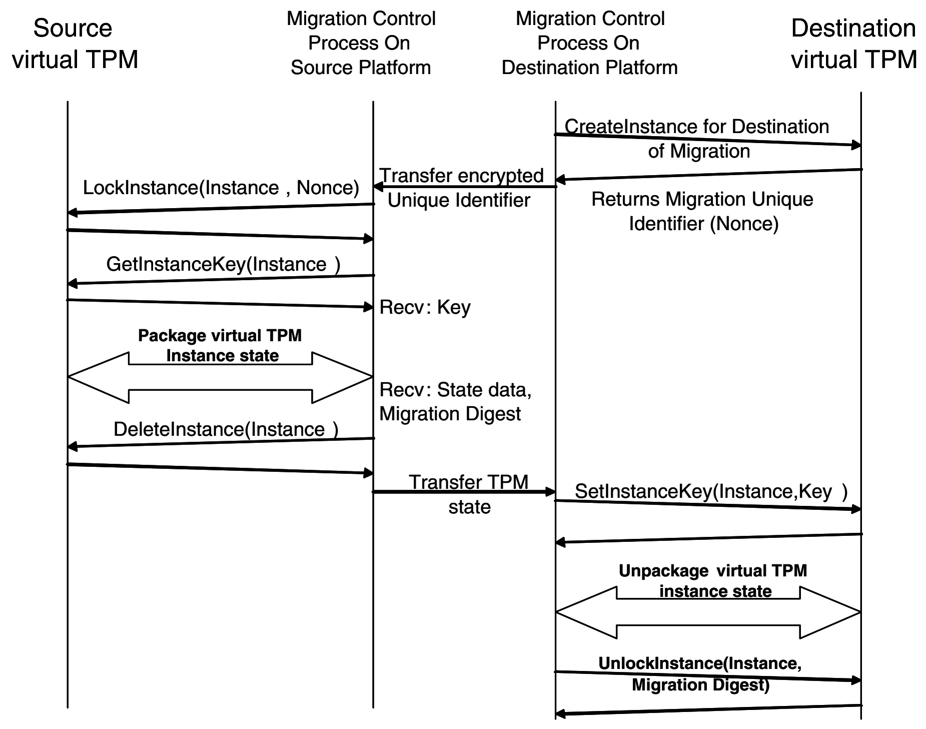

Source virtual TPM
Migration Control Process On Source Platform
Migration Control Process On Destination Platform
Destination virtual TPM
CreateInstance for Destination of Migration
Returns Migration Unique Identifier (Nonce)
Transfer encrypted Unique Identifier
LockInstance(Instance , Nonce)
GetInstanceKey(Instance)
Recv: Key
Package virtual TPM Instance state
Recv: State data, Migration Digest
DeleteInstance(Instance)
Transfer TPM state
SetInstanceKey(Instance,Key)
Unpackage virtual TPM instance state
UnlockInstance(Instance, Migration Digest)

Figure 3: Virtual TPM Migration Protocol

the vTPM manager, and other components on which the correct functioning of the virtual environment and the vTPM depends.

As previously mentioned, the certificate for a virtual TPM instance does not necessarily stand for the same security guarantees as that of a hardware TPM. If a challenger decides that the security guarantees of the virtual TPM are not sufficient he can directly challenge the virtual machine owning the hardware TPM to verify the basic measurements including the one of the virtual TPM. Section 7.2 describes how certificates can be issued to mitigate this problem.

5 Implementation

In this section we present our implementation of virtual TPM support for the Xen hypervisor [27]. We expect that an implementation for other virtualization environments would be similar in the area of virtual TPM management, but will differ in the particular management tools and device-driver structure.

We have implemented the two previously discussed solutions of a virtual TPM. One is a pure software solution targeted to run as a process in user space inside a dedicated virtual machine (Figure 1) and the other runs on IBM's PCI-X Cryptographic Coprocessor (PCIXCC) card [15] (Figure 2).

5.1 Implementation for Xen

Xen is a VMM for paravirtualized operating systems that can also support full virtualization by exploiting emerging hardware support for virtualization. In Xen-speak, each virtual machine is referred to as a *domain*. *Domain-0* is the first instance of an OS that is started during system boot. In Xen version 3.0, domain-0 owns and controls all hardware attached to the system. All other domains are *user domains* that receive access to the hardware using *frontend device drivers* that connect to *backend device drivers* in domain-0. Domain-0 effectively proxies access to hardware such as network cards or hard drive partitions.

We have implemented the following components for virtual TPM support under the Xen hypervisor:

- Split device-driver pair for connecting domains to a virtual TPM
- Scripts to help connect virtual machines to virtual TPM instances
- Virtual TPM management tools
- Virtual TPM-specific extensions to Xen's management tools (e.g., xend, xm)
- Full TPM 1.2 implementation extended with our virtual TPM command set

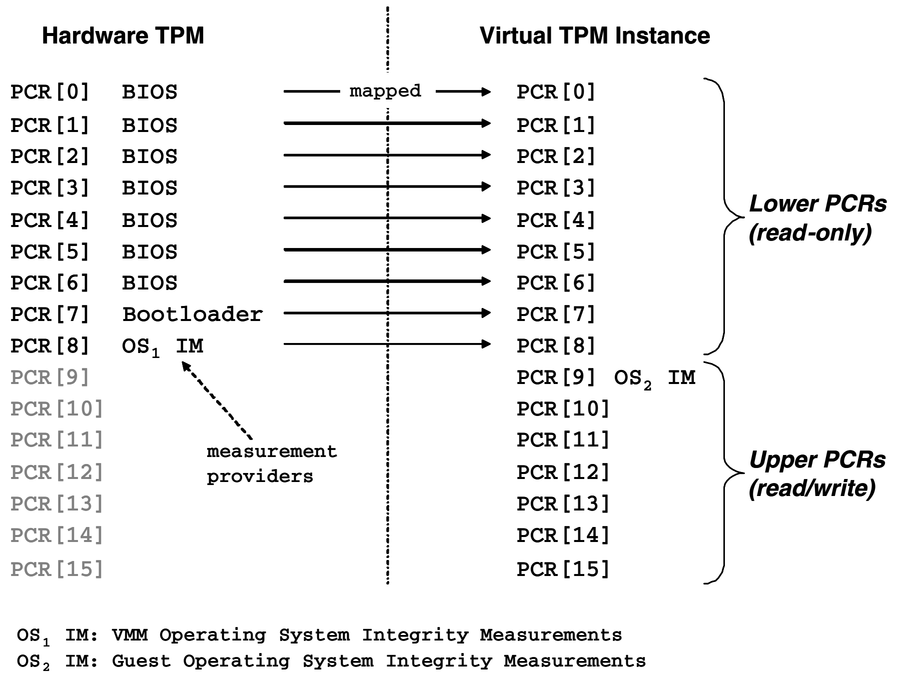


Figure 4: Mapping of Virtual TPM to Hardware TPM PCRs

We have extended the Xen hypervisor tools to support virtual TPM devices. *xm*, the Xen Management tool, parses the virtual machine configuration file and, if specified, recognizes that a virtual TPM instance must be associated with a virtual machine. *xend*, the Xen Daemon, makes entries in the *xenstore* [22] directory that indicate in which domain the TPM backend is located. Using this information, the TPM frontend driver in the user domain establishes a connection to the backend driver. During the connection phase, the backend driver triggers the Linux hotplug daemon that then launches scripts for connecting the virtual TPM instance to the domain.

Within our virtual TPM hotplug scripts, we need to differentiate whether the virtual machine was just created or whether it resumed after suspension. In the former case, we initialize the virtual TPM instance with a reset. In the latter case, we restore the state of the TPM from the time when the virtual machine was suspended. Inside the scripts we also administer a table of virtual-machine-to-virtual-TPM-instance associations and create new virtual TPM instances when no virtual TPM exists for a started virtual machine.

Figure 5 shows an example of a virtual machine configuration file with the virtual TPM option enabled. The attributes indicate in which domain the TPM backend driver is located and which TPM instance is preferred to be associated with the virtual machine. To eliminate configuration errors, the final decision on which virtual TPM instance is given to a virtual machine is made in the hotplug scripts and depends on already existing entries in the associations table.

```
kernel = "/boot/vmlinuz -2.6.12 -xen"
ramdisk = "/boot/vmlinuz -2.6.12 -xen.img"
memory = 256
name = " UserDomainWithTPM "
vtpm = ['backend=0, instance=1']
```

Figure 5: Virtual Machine Configuration File with vTPM Option

We have implemented the Xen-specific frontend driver such that it plugs into the generic TPM device driver that is already in the Linux kernel. Any application that wants to use the TPM would communicate with it through the usual device, /dev/tpm0. The backend driver is a component that only exists in the virtualized environment. There we offer a new device, /dev/vtpm, through which the virtual TPM implementation listens for requests.

Our driver pair implements devices that are connected to a Xen-specific bus for split device drivers, called *xenbus*. The xenbus interacts with the drivers by invoking their callback functions and calls the backend driver for initialization when a frontend has appeared. It also notifies the frontend driver about the request to suspend or resume operation due to suspension or resumption of the user domain.

Suspension and resumption is an important issue for our TPM frontend driver implementation. The existing TPM protocol assumes a reliable transport to the TPM hardware, and that for every request that is sent a guaranteed response will be returned. For the vTPM driver implementation this means that we need to make sure that the last outstanding response has been received by the user domain before the operating system in that domain suspends. This avoids extension of the basic TPM protocol through a more complicated sequence number-based protocol to work around lost packets.

We use Xen's existing shared memory mechanism (*grant tables* [6]) to transfer data between front- and back-end driver. Initially a page is allocated and shared between the front and back ends. When data is to be transmitted they are copied into pages and an access grant to the pages is established for the peer domain. Using an event channel, an interrupt is raised in the peer domain which then starts reading the TPM request from the page, prepends the 4-byte instance number to the request and sends it to the virtual TPM.

The virtual TPM runs as a process in user space in domain-0 and implements the command extensions we introduced in Section 4.4. For concurrent processing of requests from multiple domains, it spawns multiple threads that wait for requests on /dev/vtpm and a local interface. Internal locking mechanisms prevent multiple threads from accessing a single virtual TPM instance at the same time. Although a TPM driver implementation in a user domain should not allow more than one unanswered TPM request to be processed by a single TPM, we cannot assume that every driver is written that way. Therefore we implemented the locking mechanism as a defense against buggy TPM drivers.

The virtual TPM management tools implement command line tools for formatting and sending virtual TPM commands to the virtual TPM over its local interface. Requests are built from parameters passed through the command line. We use these tools inside the hotplug scripts for automatic management of virtual TPM instances.

5.2 Implementation for the PCI-X Cryptographic Coprocessor

IBM's PCIXCC secure coprocessor is a programmable PCI-X card that offers tamper-responsive protection. It is ideally suited for providing TPM functionality in security-sensitive environments where higher levels of assurance are required, e.g., banking and finance.

The code for the virtual TPM on the card differs only slightly from that which runs in a virtual machine. The main differences are that the vTPM on the card receives its commands through a different transport interface, and it uses built-in cryptographic hardware for acceleration of vTPM operations. To use the card in the Xen environment, a process in user space must forward requests between the TPM backend driver and the driver for the card. This is the task of the proxy in Figure 2.

Table 1 describes the properties that can be achieved for TPM functionality based on the three implementation alternatives: hardware TPM, virtual TPM in a trusted virtual machine, and virtual TPM in a secure coprocessor.

6 Sample Application

We ran an existing TPM application to show that our virtual TPM implementation provides correct TPM functionality to virtual machines. As a sample application we chose IBM's open-source Integrity Measurement Architecture (IMA) for the Linux operating system [13].

IMA provides to a remote system verifiable evidence of what software is running on a measured system. It maintains a list of hash values covering all executable content loaded into a system since startup, including application binaries. It brings together measurements made by the BIOS, boot loader and OS, and it offers an interface to retrieve these hash values from a remote system. IMA returns its list of measurements as well as a quote of current PCR values signed by the TPM. The signed quote from the TPM proves the integrity of the measurements. The remote system can then compare the measurements against known values to determine what software was loaded on the measured system.

IMA was originally written to run in a non-virtualized environment, where the Linux kernel has direct access to a hardware TPM. As a test of our vTPM facility, we ran IMA in a Xen virtual machine with access to a vTPM instance.

The complete attestation sequence in our virtualized environment is as follows. The virtual TPM runs as a process in Xen's management virtual machine, domain-0. We boot the system using a trusted boot loader, Trusted GRUB [9, 18]. We measure the Xen hypervisor executable, the domain-0 kernel and initial RAM disk, as well as the initial Xen access control policy [20], and extend a PCR in the hardware TPM with these measurements. The resulting hardware PCR value thus attests to the integrity of the vTPM's trusted computing base (TCB), namely the hypervisor plus the management virtual machine.

When a user virtual machine starts, we measure its kernel image and initial RAM disk, and extend a PCR

Properties	Implementation		
	Hardware TPM	PCIXCC	Trusted VM TPM
HW Tamper	no protection	responsive	no protection
SW Tamper	BIOS or software protected	crypto protected (signed software)	SW protected
TPM TCB	TPM chip, BIOS	tamper responsive environment, signed software	vTPM VM, hypervisor
Platform-Binding	physical (e.g., soldering)	PCI-X bus	logical (H/W TPM or BIOS)
OS-Binding	BIOS or hypervisor (if VM-owned)	hypervisor	hypervisor
Support OS	1	n	n
Cost in USD	1	$>$ 1,000	0

Table 1: Comparison of TPM Implementations

in the virtual TPM with these measurements. This sequence of measurements is part of the setup process of the vTPM instance (see Section 4.4). As the user virtual machine continues to run, the IMA-enhanced kernel in that virtual machine also extends a virtual PCR with measurements of every application that is loaded.

IMA attests to the integrity of both the vTPM TCB and the user virtual machine by returning PCR values from both the hardware TPM and the virtual TPM. We achieve this with our vTPM by projecting the lower PCRs of the hardware TPM (e.g., PCRs 0 through 8) to all virtual TPMs. This means that if a user VM reads one of those PCRs, the vTPM facility actually fetches the value from the hardware TPM. Extending hardware PCRs from user VMs is therefore disabled since these registers are logically owned by the management VM as depicted in Figure 4. Upper PCRs are accessible by user VMs as usual.

Therefore, we have the management VM extend the lower PCRs with measurements of the vTPM TCB. We have the user VM extend the upper PCRs with measurements of the user VM itself. IMA reports then combine lower PCR values, higher PCR values, and the measurement list from both the user VM and the management VM to provide a comprehensive view of the system. To relay the names of applications measured into the hardware TPM, we implemented a small extension to Integrity Measurement Architecture that retrieves this information from the vTPM-hosting domain using the ReportEnvironment command. Other aspects of IMA were left unmodified.

7 Discussion and Future Work

In section 3 we introduced the requirements that an architecture for enabling TPM support in a virtual environment must fulfill. So far we have presented solutions for the first three items and described their implementations. We will now revisit our initial requirements and compare them against our implementation. We then discuss solutions for the remaining items.

7.1 Requirements Revisited

Unmodified TPM Usage Model We provide TPM support by emulating device functionality through a software implementation. We designed and implemented an architecture for the Xen hypervisor that enables us to connect each user domain to its own TPM instance. With our command set extensions we can create as many virtual TPM instances as needed. All existing TPM V 1.2 commands are available to a user domain and the TPM command format remains unchanged.

Strong Virtual Machine to Virtual TPM Association We have shown a design that supports strong virtual machine to virtual TPM association. Components that enforce this need to be implemented in the backend driver such that TPM packets can be routed to the appropriate domain. Also, a table of virtual machine to virtual TPM instance must be maintained.

We introduced new TPM commands for secure migration of TPM state between two virtual TPM implementations. Our migration protocol guarantees TPM uniqueness and prevents attacks on the TPM state information such as alterations to or omission of pieces of state information. We based virtual TPM migration on TPM key migration.

In our design we assume the trustworthiness of the destination TPM implementation and the uniqueness of migration identifiers (which can all be verified). HMAC values and migration digests are verified such that our security features can be enforced. It is important for virtual TPM migration that the asymmetric storage key is only migrated into a trusted virtual TPM. A possible solution for determining the trustworthiness of a destination TPM is to require a certificate of the destination TPM's storage key where the signature key is an externally certified Attestation Identity Key (AIK).

Strong Association of the Virtual TPM with the Underlying TCB Using an existing attestation architecture for Linux, we showed how a strong association between a virtual TPM instance and the hardware root of trust (hardware TPM) of the platform can be established.

Our architecture and virtual TPM have been designed such that a challenger not only sees measurements taken inside the virtual machine OS, but can establish trust into the virtualization environment, including the boot process, hypervisor and the operating system that is hosting the virtual TPM.

7.2 Trust Establishment

We have so far reported several solutions from our experience providing TPM support to virtual machines. However, there are a number of issues that still need to be investigated. Whereas other devices can be satisfactorily virtualized through device emulation, more support is needed in our case, particularly on the treatment of security credentials such as TPM keys and associated certificates.

From our experience we can claim that it is easy to create an endorsement key for a virtual TPM instance, but some questions arise around the certificate that needs to be issued:

- Who would provide a certificate for an endorsement key of a virtual TPM?
- What guarantees would this certificate stand for?

A certificate authority, i.e., a privacy CA, bases its decision to certify an AIK of a TPM on the certificate of the EK that a manufacturer provides along with the device. This certificate vouches for the TPM being a hardware device and that it is firmly attached to the motherboard of the computer. Since the availability of an EK certificate plays this important role in receiving a certificate for AIKs, the EK certificate should also be available to a virtual TPM instance even if it does not stand for the

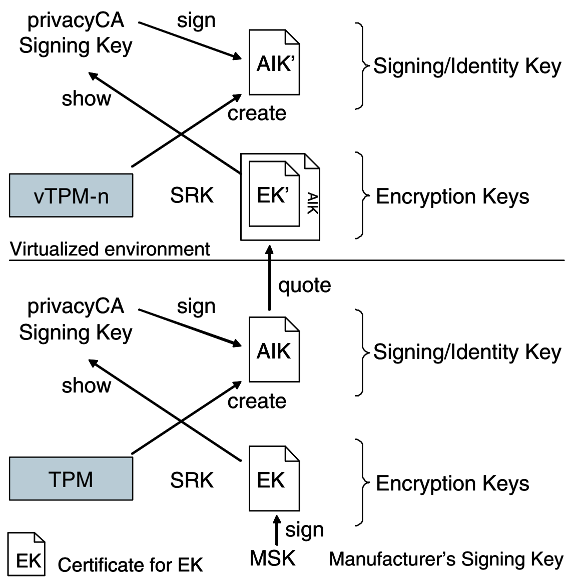


Figure 6: Certification of Endorsement Key using an AIK

same security guarantees as those provided by a hardware TPM. However, virtual TPMs can be dynamically created whenever a new VM is created, and therefore requests for EK certificates can become more frequent and their management becomes much more dynamic.

We have found several solutions for the creation of EK certificates, each having advantages and disadvantages. We discuss those solutions below and, after looking at virtual TPM migration, provide a comparison between them.

1. Our first solution creates a certificate chain by connecting the certificate issued for the EK of a virtual TPM instance to that of an AIK of the hardware TPM. Figure 6 depicts this relationship. It shows that a privacy CA issues certificates for AIKs of a virtual TPM based on the certificate of its endorsement key EK'. The advantage of this scheme is that we have preserved the normal procedure of acquiring an AIK' certificate by submitting the certificate of EK' to a privacy CA for evaluation.

   In this and the following solutions we are using an (attestation) identity key and the TPM's Quote command to issue a signature over the current state of PCRs and a user-provided 160bit number. We provide as 160bit number the SHA1 hash of the certificate contents of the EK'. The resulting signature ties this EK' certificate and the virtual TPM instance to the underlying platform. In addition to the PCRs, the certificate can also contain the measurement list of the VM environment to enable the establishment

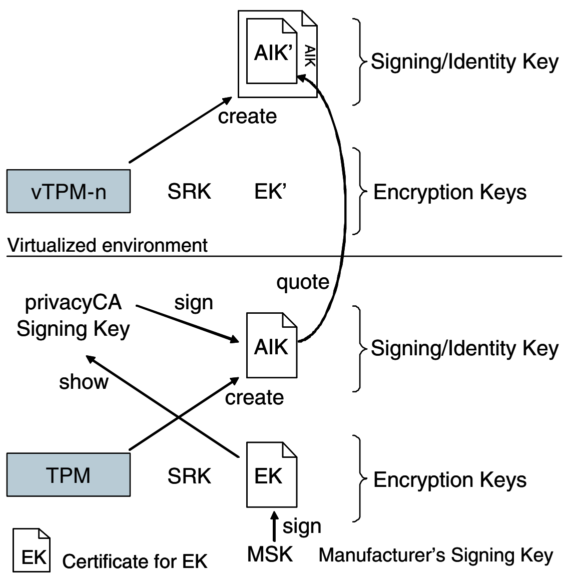


Figure 7: Certification of AIK' using AIK

of trust into the certificate-signing process [21].

2. Our second solution, depicted in Figure 7, does not use a certificate for a virtual EK', but issues certificates for virtual TPM AIKs based on an AIK issued for the hardware TPM. The resulting certificate chain ties the virtual TPM's AIK' to the AIK of the hardware TPM, and thus to the hosting system. The advantage of this solution is that once an AIK has been issued for the hardware TPM, virtual TPM AIKs for guest VMs can also be quickly certified. Through the chain, a link is established to the hardware-TPM platform. The disadvantage of this solution is that it requires changes to the normal procedure of acquiring an AIK certificate from a certificate authority.

3. A third solution relies on a local authority to issue the certificate for the virtual TPM instance's EKs. The benefit of this procedure compared to the previous ones is that the resulting virtual TPM's EK certificate is not tied to the hardware platform, since no certificate chain is established to credentials of the hardware TPM. A local EK certificate authority can also be used for hardware TPMs if they are not equipped with a platform certificate, as is often the case today. Beyond this, this third solution offers the advantage over the second one of not changing the procedure for acquiring certificates for AIKs.

4. A fourth solution is based on a secure coprocessor that replaces the hardware TPM used in the other solutions to provide a hardware root of trust. The manufacturer links the endorsement key certificate to the secure coprocessor certificate and enables remote parties to establish security properties for the virtual TPM runtime environment as described in Section 5.2.
   Starting with this manufacturer-provided EK certificate, all the previously described solutions for creating certificate chains for virtual TPM instances can be applied.

Depending on which solution for issuing certificates is chosen, the migration of a virtual TPM to another platform can affect the validity of certified TPM keys. If, for example, AIKs have been certified based on an EK that was previously tied to the hardware platform through a chain, as we have shown in the first two solutions, the AIKs must be invalidated once the VM is resumed on the target platform since the link to the old platform has now been broken. Our third solution avoids this problem, because it does not establish a firm link with the VM-hosting platform.

What makes the realization of an architecture based on certificate chains more difficult is that AIKs and certificates may be maintained by programs inside the operating system. The TSS stack must be aware of migration and destroy AIKs once the OS resumes on the target platform. After the AIKs have been recreated, they must be certified for usage on the new platform. Applications must also be made aware of the new certificates and remove old ones from memory.

Another problem can be certificates that clients examine while a VM is migrating to a new platform. Based on the evaluation of the certificate, the client may treat the peer system as trusted, although it is now running inside a new environment. For practical purposes, a migrating partition should offer a subscription service for any party interested in learning about migration. Notifications can be sent that inform subscribers that migration has happened and trigger a reestablishment of trust. We do not currently offer such a service.

Another question that arises due to virtual machine migration is: When a virtual machine is migrated from one system to another, should all virtual machine environments' measurements be recorded and a history be established? We feel the answer to this question is ”yes”, but we have not yet explored efficient ways to support this capability.

Table 2 gives an overview of the properties of the first three of our proposed solutions. A decision about which method to implement for certifying EKs must weigh the advantages and disadvantages of each solution. If a

Support \ Method	AIK certifies EK'	AIK certifies AIK'	Local authority certifies EK'
Needs a CA	privacy CA	No	local and privacy CA
Establishes link to hosting platform	Yes, EK' is linked	Yes, AIK' is linked	No
Needs AIKs and certificates to be invalidated after migration	Yes; must also invalidate EK	Yes	No
How external party verifies AIK' certificate	Need to know public signing key of privacy CA	Need to know public AIK used for quoting	Need to know public signing key of privacy CA
Software in VM needs to be aware of how to have AIK' certified	No - contact privacy CA as normal	Yes. AIK of parent environment is used to have AIK' certified.	No - contact privacy CA as normal
Credentials a CA must interpret to issue AIK certificate	AIK: EK certificate AIK': TPM-quote for EK' and associated public AIK	AIK: EK certificate AIK': not involved	AIK: EK certificate AIK': EK' certificate and (local) CA's public signing key

Table 2: Comparison of Methods to Issue Certificates for AIKs

strong connection between the virtual TPM and the hardware TPM is desired, then one of solution 1,2 or 4 should be implemented. However, it will be necessary in this case to invalidate the chained certificates and keys after migration in order to reestablish a chain to the new hardware root of trust. In that respect our second solution offers better support for a dynamic environment, since here only the AIKs of the virtualized environment need to be recreated and certified. The first solution would eventually have to place the EK' certificate on a revocation list and create a new EK.

If a local certification process has already been established to certify EKs for hardware TPMs, this or a similar process can be applied to EKs of virtual TPMs as well. It would simplify an implementation for virtual TPM migration with its VM since in this case there is no link to the parent environment. Therefore migration would not break any certificate chains. It can be regarded as the least complicated solution, since neither side of the attestation procedure would have to forget about credentials that applied to the pre-migration environment.

8 Related Work

The Xen open-source repository [27] contains a limited virtual TPM implementation comprised of combined contributions by Intel Corporation and the authors of this paper. Our contributions to Xen so far include the virtual TPM driver pair (front- and back-end drivers), hotplug scripts, and changes to Xen's management tools. We kept this infrastructure modular so that different realizations of virtual TPMs can work with it. The virtual TPM design and implementation presented in this paper adds the following to what is currently available in Xen: support for migrating a vTPM instance alongside its associated virtual machine, support for attestation of the complete vTPM environment along with the contents of a virtual machine, and an entirely separate software implementation of the TPM specification. In addition, the virtual TPM now in Xen is a partial implementation based on version 1.1 of the TPM specification, while we have updated our virtual TPM to be a complete implementation of version 1.2.

Previous research in the area of trusted computing examined how data that is protected (sealed) by a hardware TPM can be moved to another platform. Kuehn et al. [17]

proposed a protocol for migrating the key-related hardware TPM security state from one hardware platform to another involving a separate *TPM Migration Authority* (TMA). Our protocol differs from the one presented there in many significant ways. Most notably, we migrate the complete virtual TPM state, we do not require a third party for migration, we maintain associations of virtual TPMs to their VMs and the operating system, and we can seamlessly integrate our protocol into the automated VM migration process. In addition, the extensions we introduce to the TPM standard do not require changes to existing commands and semantics. Similar to their concern about security of the destination TPM, we have pointed out that secure migration relies on a decision process that determines the safety of migrating a key pair to another TPM based on trust in that other TPM implementation.

The Terra project [7] investigated trusted virtual machine monitors. They developed a prototype based on VMWare's GSX server product that performs attestation of virtual machines and applications launched therein. Their publications recognize the availability of TPM 1.1b, but do not describe the design of a virtual TPM to run their attestation scheme against. Terra could use something like our vTPM facility to make a virtual TPM instance available to each of their virtual machines.

9 Conclusions

We have designed and implemented a system that provides trusted computing functionality to every virtual machine on a virtualized hardware platform. We virtualized the Trusted Platform Module by extending the standard TPM command set to support vTPM lifecycle management and enable trust establishment in the virtualized environment. We added support for secure vTPM migration while maintaining a strong association between a vTPM instance and its associated VM.

We uncovered the most important difficulties that arise when virtualizing the TPM. Whereas usually virtualization of hardware devices can be achieved through software emulation, we have demonstrated that this is not sufficient in the case of the TPM. Certificates that may exist for hardware TPMs and vouch for strong security properties need to be issued for virtual TPM instances' endorsement keys . These certificates can naturally not represent the same properties for a virtual TPM process running in user space. Trust chains that are usually owned by a single OS now *pass through* a hierarchy of virtual machines. Virtual TPM migration can create further problems if certificate chains that have been established break or trust must be reestablished.

We virtualized the Trusted Platform Module by making all low-level TPM 1.2 commands available to every virtual machine. Applications that don't handle certificates related to TPM-generated keys or do not deal with the concept of trust can remain unchanged. Applications challenging a virtual machine or those following certificate chains, like for example a privacy CA, must be aware of the modifications that were necessary for the virtualized environment. Those modifications include certificate chains that consist of different types of certificates issued through special signing mechanisms of the virtual TPM, or certificates provided by the manufacturer of the device or those issued through a certificate authority such as a privacy CA. Applications that have been adapted to work in the virtualized environment will be backwards compatible with platforms using a singleton hardware TPM.

Our proposed architecture for virtualizing the TPM is a major building block for establishing trust in virtualized environments. For example, Trusted Virtual Data Centers [16] create distributed virtual domains offering strong enterprise-level security guarantees in hosted data center environments. In such an environment, virtual TPMs help to establish trust in strong domain security guarantees through their remote attestation and sealing capabilities.

Acknowledgments

We would like to thank our shepherd, Peter Chen, and the anonymous reviewers for their valuable comments. We also thank the Xen community for their feedback on the vTPM work.

References

[1] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[2] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI '05)*, 2005.

[3] Common Criteria. Trusted Computing Group (TCG) Personal Computer (PC) Specific Trusted Building Block (TBB) Protection Profile and TCG PC Specific TBB With Maintenance Protection Profile, July 2004.

[4] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[5] R. Figueiredo, P. A. Dinda, and J. Fortes. Resource virtualization renaissance. *IEEE Computer*, 38(5):28–31, 2005.

[6] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *Proceedings of the OASIS ASPLOS Workshop*, 2004.

[7] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a Virtual Machine-based Platform for Trusted Computing. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 193–206, 2003.

[8] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer Magazine*, 7(6):34–45, 1974.

[9] Applied Data Security Group. What is TrustedGRUB, http://www.prosec.ruhr-uni-bochum.de/trusted_grub.html.

[10] Trusted Computing Group. TCG TPM Specification Version 1.2 - Part 1 Design Principles, 2005.

[11] Trusted Computing Group. TCG TPM Specification Version 1.2 - Part 3 Commands, 2005.

[12] IBM. IBM eServer x366. http://www-03.ibm.com/servers/eserver/xseries/x366.html.

[13] IBM. Integrity Measurement Architecture for Linux. http://www.sourceforge.net/projects/linux-ima.

[14] IBM. PHYP: Converged POWER Hypervisor Firmware for pSeries and iSeries. http://www-1.ibm.com/servers/enable/site/peducation/abstracts/abs_2bb2.html.

[15] IBM. Secure Coprocessing. http://www.research.ibm.com/secure_systems_department/projects/scop/index.html.

[16] IBM. Trusted Virtual Data Center. http://domino.research.ibm.com/comm/research_projects.nsf/pages/ssd_trustedvirtualdatacenter.index.html.

[17] U. Kühn, K. Kursawe, S. Lucks, A. Sadeghi, and C. Stüble. Secure data management in trusted computing. In *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES 2005)*, 2005.

[18] H. Maruyama, F. Seliger, N. Nagaratnam, T. Ebringer, S. Munetoh, S. Yoshihama, and T. Nakamura. Trusted platform on demand. Technical Report RT0564, IBM, February 2004.

[19] National Institute of Standards and Technology. Secure Hash Standard (SHA-1). Federal Information Processing Standards Publication 180-1, 1993.

[20] R. Sailer, T. Jaeger, E. Valdez, R. Cáceres, R. Perez, S. Berger, J. Griffin, and L. van Doorn. Building a MAC-based security architecture for the Xen opensource hypervisor. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, December 2005.

[21] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the USENIX Security Symposium*, 2004.

[22] The Xen Team. Xen Interface Manual - Xen v3.0 for x86.

[23] Trusted Computing Group. http://www.trustedcomputinggroup.org.

[24] Trusted Computing Group. TCG PC Specific Implementation Specification, 2003.

[25] Trusted Computing Group. TCG Software Stack (TSS) Specification - Version 1.10 Golden, 2003.

[26] VMware, Inc. http://www.vmware.com.

[27] Xensource. Xen Open-Source Hypervisor. http://www.xensource.com/products/downloads.

Designing voting machines for verification

NAVEEN SASTRY[*] TADAYOSHI KOHNO[†] DAVID WAGNER[‡]

Abstract

We provide techniques to help vendors, independent testing agencies, and others verify critical security properties in direct recording electronic (DRE) voting machines. We rely on specific hardware functionality, isolation, and architectural decision to allow one to easily verify these critical security properties; we believe our techniques will help us verify other properties as well. Verification of these security properties is one step towards a fully verified voting machine, and helps the public gain confidence in a critical tool for democracy. We present a voting system design and discuss our experience building a prototype implementation based on the design in Java and C.

1 Introduction

With a recent flurry of reports criticizing the trustworthiness of direct recording electronic (DRE) voting machines, computer scientists have not been able to allay voters' concerns about this critical infrastructure [17, 29, 33, 38]. The problems are manifold: poor use of cryptography, buffer overflows, and in at least one study, poorly commented code. Given these problems, how can we reason about, or even prove, security properties of voting machines?

The ultimate security goal would be a system where any voter, without any special training, could easily convince themselves about the correctness of *all* relevant security properties. Our goal is not so ambitious; we address convincing those with the ability to understand code the correctness of a few security properties. For clarity, we focus on two important security properties in the body of this paper. Verification of these properties, as well as the others we describe elsewhere in this paper, are a step towards the full verification of a voting machine.

Property 1 *None of a voter's interactions with the voting machine, including the final ballot, can affect any subsequent voter's sessions*[1].

One way to understand this property is to consider a particular voting system design that exhibits the property. A DRE can be "memoryless," so that after indelibly storing the ballot, it erases all traces of the voter's actions from its RAM. This way, a DRE cannot use the voter's choices in making future decisions. A DRE that achieves Property 1 will prevent two large classes of attacks: one against election integrity and another against privacy. A DRE that is memoryless cannot decide to change its behavior in the afternoon on election day if it sees the election trending unfavorably for one candidate. Similarly, successful verification of this property guarantees that a voter, possibly with the help of the DRE or election insider, cannot access a prior voter's selections.

A second property is:

Property 2 *A ballot cannot be cast without the voter's consent to cast.*

Property 2 ensures the voter's ballot is only cast with their consent; combined with other security properties, the property helps ensure the voter's ballot is cast in an unmodified form.

In Section 8, we discuss additional target properties for our architecture, and we discuss strategies for how to prove and implement those properties successfully.

[*]`nks@cs.berkeley.edu`. Supported by NSF CNS-0524252 and by the Knight Foundation under a subcontract through the Caltech/MIT Voting Technology Project.

[†]`tkohno@cs.ucsd.edu`. Supported by NSF CCR-0208842, NSF ANR-0129617, and NSF CCR-0093337. Part of this research was performed while visiting the University of California at Berkeley.

[‡]`daw@cs.berkeley.edu`. Supported by NSF CCR-0093337 and CNS-0524252.

[1]Note that we do allow certain unavoidable interactions, e.g., after the ballot storage device becomes "full," a voting machine should not allow subsequent voters to vote.

Current DREs are not amenable to verification of these security properties; for instance, version 4.3.1 of the Diebold AccuVote-TS electronic voting machine consists of 34 712[2] lines of vendor-written C++ source code, all of which must be analyzed to ensure Properties 1 and 2. One problem with current DRE systems, in other words, is that the trusted computing base (TCB) is simply too large. The larger problem, however, is the code simply is not *structured* to verify security properties.

In this paper, we develop a new architecture that significantly reduces the size of the TCB for verification of these properties. Our goal is to make voting systems more amenable to efficient verification, meaning that implementations can be verified to be free of malicious logic. By appropriate architecture design, we reduce the amount of code that would need to be verified (e.g., using formal methods) or otherwise audited (e.g., in an informal line-by-line source code review) before we can trust the software, thereby enhancing our ability to gain confidence in the software. We stress that our architecture assumes voters will be diligent: we assume that each voter will closely monitor their interaction with the voting machines and look for anomalous behavior, checking (for example) that her chosen candidate appears in the confirmation page.

We present techniques that we believe are applicable to DREs. We develop a partial voting system, but we emphasize that this work is not complete. As we discuss in Section 2, voting systems comprise many different steps and procedures: pre-voting, ballot preparation, audit trail management, post-election, recounts, and an associated set of safeguard procedures. Our system only addresses the active voting phase. As such, we do *not* claim that our system is a replacement for an existing DRE or a DRE system with a paper audit trail system. See Section 7 for a discussion of using paper trails with our architecture.

Technical elements of our approach. We highlight two of the key ideas behind our approach. First, we focus on creating a trustworthy vote confirmation process. Most machines today divide the voting process into two phases: an initial vote selection process, where the voter indicates who they wish to vote for; and a vote confirmation process, where the voter is shown a summary screen listing their selections and given an opportunity to review and confirm these selections before casting their ballot. The vote selection code is potentially the most complex part of the system, due to the need for complex user interface logic. However, if the confirmation process is easy to verify, we can verify many important security properties without analyzing the vote selection process. Our architecture splits the vote confirmation code into a separate module whose integrity is protected using hardware isolation techniques. This simple idea greatly reduces the size of the TCB and means that only the vote confirmation logic (but not the vote selection logic) needs to be examined during a code review for many security properties, such as Property 2.

Second, we use hardware resets to help ensure Property 1. In our architecture, most modules are designed to be stateless; when two voters vote in succession, their execution should be independent. We use hard resets to restore the state of these components to a consistent initial value between voters, eliminating the risk of privacy breaches and ensuring that all voters are treated equally by the machine.

Our architecture provides several benefits. It preserves the voting experience that voters are used to with current DREs. It is compatible with accessibility features, such as audio interfaces for voters with visual impairments, though we stress that we do not implement such features in our prototype. It can be easily combined with a voter-verified paper audit trail (VVPAT). Our prototype implementation contains only 5 085 lines of trusted code.

2 Voting overview

DREs. A *direct recording electronic* (DRE) voting machine is typically a stand-alone device with storage, a processor, and a computer screen that presents a voter with election choices and records their selections so they can be counted as part of the canvass. These devices often use an LCD and touch screen to interact with the voter. Visually impaired voters can generally use alternate input and output methods, which presents a boon to some voters who previously required assistance to vote.

Pre-election setup. The full election process incorporates many activities beyond what a voter typically experiences in the voting booth. Although the exact processes differ depending on the specific voting technology in question, Figure 1 overviews the common steps for DRE-based voting. In the pre-voting stage, election officials prepare ballot definition files describing the parameters of the election. Ballot definition files can be very complex [24], containing not only a list of races and values indicating how many selections a voter can make for each race, but also containing copies of the ballots in multiple languages, audio tracks for visually impaired voters (possibly also in multiple languages), fields that vary by precinct, and fields that vary by the voter's party affiliation for use in primaries. Election officials generally use external software to help them generate the ballot definition files. After creating the ballot definition files, an election worker will load those files onto the DRE vot-

[2]Kohno et al. count the total number of lines in their paper [17]; for a fair comparison with our work, we look at source lines of code, which excludes comments and whitespace from the final number. Hence, the numbers cited in their paper differ from the figure we list.

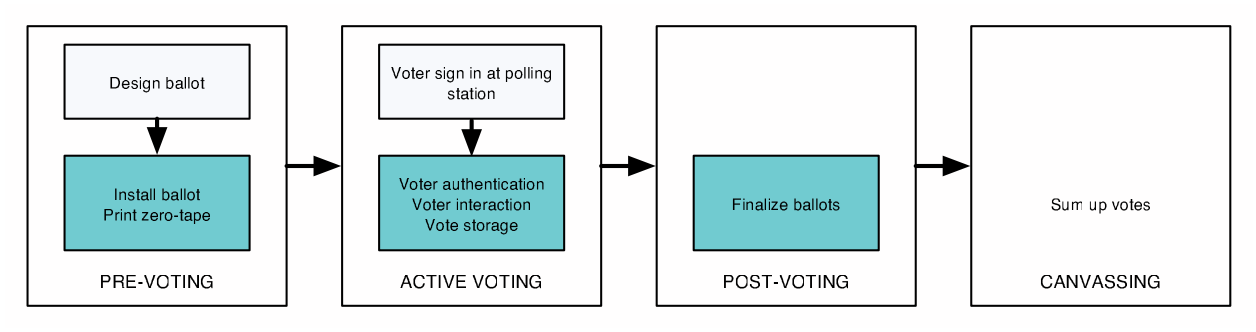


Figure 1: Major steps in the voting process when using DREs. The shaded portions are internal to the DREs. In this work, we mainly address voter authentication, interaction, and vote storage.

ing machines. Before polls open, election officials generally print a "zero tape," which shows that no one cast a ballot prior to the start of the election.

Active voting. When a voter Alice wishes to vote, she must first interact with election officials to prove that she is eligible to vote. The election officials then give her some token or mechanism to allow her to authenticate herself to the DRE as an authorized voter. Once the DRE verifies the token, the DRE displays the ballot information appropriate for Alice, e.g., the ballot might be in Alice's native language or, for primaries, be tailored to Alice's party affiliation. After Alice selects the candidates she wishes to vote for, the DRE displays a "confirmation screen" summarizing Alice's selections. Alice can then either accept the list and cast her ballot, or reject it and return to editing her selections. Once she approves her ballot, the DRE stores the votes onto durable storage and invalidates her token so that she cannot vote again.

Finalization & post-voting. When the polls are closed, the DRE ensures that no further votes can be cast and then prints a "summary tape," containing an unofficial tally of the number of votes for each candidate. Poll workers then transport the removable storage medium containing cast ballot images, along with the zero tape, summary tape, and other materials, to a central facility for tallying. During the canvass, election officials accumulate vote totals and cross-check the consistency of all these records.

Additional steps. In addition to the main steps above, election officials can employ various auditing and testing procedures to check for malicious behavior. For example, some jurisdictions use parallel testing, which involves sequestering a few machines, entering a known set of ballots, and checking whether the final tally matches the expected tally. Also, one could envision repeating the vote-tallying process with a third-party tallying application, although we are unaware of any instance where this particular measure has been used in practice. While these additional steps can help detect problems, they are by no means sufficient.

3 Goals and assumptions

Security goals. For clarity, in the body of this paper we focus on enabling efficient verification of Properties 1 and 2 (see Section 1), though we hope to enable the efficient verification of other properties as well. Property 1 reflects a privacy goal: an adversary should not be able to learn any information about how a voter voted besides what is revealed by the published election totals. Property 2 reflects an integrity goal: even in the presence of an adversary, the DRE should record the voter's vote exactly as the voter wishes. Further, an adversary should not be able to undetectably alter the vote once it is stored. We wish to preserve these properties against the classes of adversaries discussed below.

Wholesale and retail attacks. A wholesale attack is one that, when mounted, has the potential of affecting a broad number of deployed DREs. A classic example might be a software engineer at a major DRE manufacturer inserting malicious logic into her company's DRE software. Prior work has provided evidence that this it is a concern for real elections [3]. Such an attack could have nationwide impact and could compromise the integrity of entire elections, if not detected. Protecting against such wholesale attacks is one of our primary goals. In contrast, a retail attack is one restricted to a small number of DREs or a particular polling location. A classic retail attack might be a poll worker stuffing ballots in a paper election, or selectively spoiling ballots for specific candidates.

Classes of adversaries. We desire a voting system that:

- Protects against *wholesale* attacks by election officials, vendors, and other insiders.

- Protects against *retail* attacks by insiders when the attacks *do not* involve compromising the physical security of the DRE or the polling place (e.g., by modifying the hardware or software in the DRE or tampering with its surrounding environment).

- Protects against attacks by outsiders, e.g., voters, when the attacks *do not* involve compromising physical security.

We explicitly do not consider the following possible goals:

- Protect against *retail* attacks by election insiders and vendors when the attacks *do* involve compromising physical security.

- Protect against attacks by outsiders, e.g., voters, when the attacks *do* involve compromising physical security.

On the adversaries that we explicitly do not consider. We explicitly exclude the last two adversaries above because we believe that adversaries who can violate the physical security of the DRE will always be able to subvert the operation of that DRE, no matter how it is designed or implemented. Also, we are less concerned about physical attacks by outsiders because they are typically *retail attacks*: they require modifying each individual voting machine one-by-one, which is not practical to do on a large scale. For example, to attack privacy, a poll worker could mount a camera in the voting booth or, more challenging but still conceivable, an outsider could use Tempest technologies to infer a voter's vote from electromagnetic emissions [18, 37]. To attack the integrity of the voting process, a poll worker with enough resources could replace an entire DRE with a DRE of her own. Since this attack is possible, we also do not try to protect against a poll worker that might selectively replace internal components in a DRE. We assume election officials have deployed adequate physical security to defend against these attacks.

We assume that operating procedures are adequate to prevent unauthorized modifications to the voting machine's hardware or software. Consequently, the problem we consider is how to ensure that the original design and implementation are secure. While patches and upgrades to the voting system firmware and software may occasionally be necessary, we do not consider how to securely distribute software, firmware, and patches, nor do we consider version control between components.

Attentive voters. We assume that voters are attentive. We require voters to check that the votes shown on the confirmation screen do indeed accurately reflect their intentions; otherwise, we will not be able to make any guarantees about whether the voter's ballot is cast as intended. Despite our reliance on this assumption, we realize it may not hold for all people. Voters are fallible and not all will properly verify their choices. To put it another way, our system offers voters the *opportunity* to verify their vote. If voters do not take advantage of this opportunity, we cannot help them. We do not assume that all voters will avail themselves of this opportunity, but we try to ensure that those who do, are protected.

4 Architecture

We focus this paper on our design and implementation of the "active voting" phase of the election process (cf. Figure 1). We choose to focus on this step because we believe it to be one of the most crucial and challenging part of the election, requiring interaction with voters and the ability to ensure the integrity and privacy of their votes. We remark that we attempt to reduce the trust in the canvassing phase by designing a DRE whose output record is both privacy-preserving (anonymized) and integrity-protected.

4.1 Architecture motivations

To see how specific design changes to traditional voting architectures can help verify properties, we will go through a series of design exercises starting from current DRE architectures and finishing at our design. The exercises will be motivated by trying to design a system that clearly exhibits Properties 1 and 2.

Resetting for independence. A traditional DRE, for example the Diebold AccuVote-TS, is designed as a single process. The functions of the DRE—validating the voter, presenting choices, confirming those choices, storing the ballot, and administrative functions—are all a part of the same address space.

Let us examine one particular strategy we can use to better verify Property 1 ("memorylessness"), which requires that one voter's selections must not influence the voting experience observed by the next voter. Suppose after every voter has voted, the voting machine is turned off and then restarted. This is enough to ensure that the voting machine's memory will not contain any information about the prior voter's selections when it starts up. Of course, the prior voter's selections must still be recorded on permanent storage (e.g., on disk) for later counting, so we also need some mechanism to prevent the machine from reading the contents of that storage medium. One conservative strategy would be to simply require that any file the DRE writes to must always be

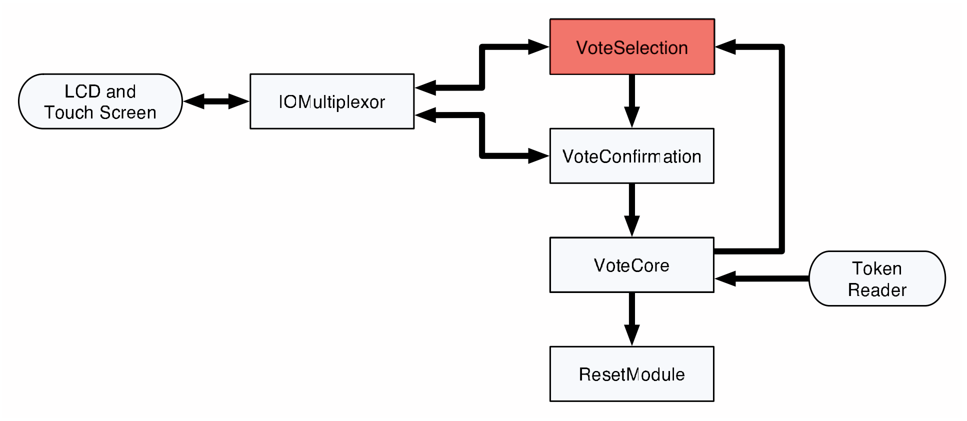


Figure 2: Our architecture, at an abstract level. For the properties we consider, the VoteSelection module need not be trusted, so it is colored red.

opened in write-only mode, and should never be opened for reading. More generally, we can allow the DRE to read from some files, such as configuration files, as long as the DRE does not have the ability to write to them. Thus the set of files on permanent storage are partitioned into two classes: a set of read-only files (which cannot be modified by the DRE), and a set of write-only files (which cannot be read by the DRE). To summarize, our strategy for enforcing Property 1 involves two prongs:

1. Ensure that a reboot is always triggered after a voter ends their session.
2. Check every place a file can be opened to ensure that data files are write-only, and configuration files are read-only.

There must still be a mechanism to prevent the DRE from overwriting existing data, even if it cannot read that data.

We introduce a separate component whose sole job is to manage the reset process. The BallotBox triggers the ResetModule after a ballot is stored. The reset module then reboots a large portion of the DRE and manages the startup process. We use a separate component so that it is simple to audit the correctness of the ResetModule.

We emphasize this design strategy is not the only way to verify this particular property. Rather, it is one technique we can implement that reduces the problem of enforcing Property 1 to the problem of enforcing a checklist of easier-to-verify conditions that suffice to ensure Property 1 will always hold.

Isolation of confirmation process. In considering Property 2, which requires the voter's consent to cast in order for the ballot to be stored, we will again see how modifying the DRE's architecture in specific ways can help verify correctness of this property.

The consent property in consideration requires auditors to confidently reason about the casting procedures. An auditor (perhaps using program analysis tools) may have an easier time reasoning about the casting process if it is isolated from the rest of the voting process. In our architecture, we take this approach in combining the casting and confirmation process, while isolating it from the vote selection functionality of the DRE. With a careful design, we only need to consider this sub-portion to verify Property 2.

From our DRE design in the previous section, we introduce a new component, called the VoteConfirmation module. With this change, the voter first interacts with a VoteSelection module that presents the ballot choices. After making their selections, control flow passes to the VoteConfirmation module that performs a limited role: presenting the voter's prior selections and then waiting for the voter to either 1) choose to modify their selections, or 2) choose to cast their ballot. Since the VoteConfirmation module has limited functionality, it only needs limited support for GUI code; as we show in Section 6.1 we can more easily analyze its correctness since its scope is limited. If the voter decides to modify the ballot, control returns to the VoteSelection module.

Note the voter interacts with two separate components: first the VoteSelection component and then VoteConfirmation. There are two ways to mediate the voter's interactions with the two components: 1) endow each component with its own I/O system and screen; 2) use one I/O system and a trusted I/O "multiplexor" to manage which component can access the screen at a time. The latter approach has a number of favorable features. Perhaps the most important is that it preserves the voter's experience as provided by existing DRE systems. A voting machine with two screens requires voters to change their voting patterns, and can introduce the opportunity for confusion or even security vulnerabilities. Another advantage is cost: a second screen adds cost and complexity. One downside is that we must now verify properties about the IOMultiplexor. For example, it must route the input and output to the proper module at the appropriate times.

In the the final piece of our architecture, we introduce a VoteCore component. After the voter interacts with the VoteSelection system and then the VoteConfirmation module to approve their selection, the VoteCore component stores the ballot on indelible storage in its BallotBox and then cancels the voter's authentication token. Then, as we described above, it initiates a reset with the ResetModule to clear the state of all modules.

Let us return to our original property: how can we verify that a ballot can only be cast with the voter's approval? With our architecture, it suffices to verify that:

1. A ballot can only enter the VoteCore through the VoteConfirmation module.

2. The VoteCore gives the voter the opportunity to review the exact contents of the ballot.

3. A ballot can only be cast if the voter unambiguously signals their intent to cast.

To prove the last condition, we add hardware to simplify an auditor's understanding of the system, as well as to avoid calibration issues with the touch screen interface. A physical cast button, enabled only by the confirmation module, acts as a gate to stop the ballot between the VoteSelection and VoteCore modules. The software in the VoteConfirmation module does not send the ballot to the VoteCore until the CastButton is depressed; and, since it is enabled only in the VoteConfirmation module, it is easy to gain assurance that the ballot cannot be cast without the voter's consent. Section 6.1 will show how we achieve this property based on the code and architecture.

There is a danger if we must adjust the system's architecture to meet each particular security property: a design meeting all security properties may be too complex. However, in Section 8, we discuss other security properties and sketch how we can verify them *with the current architecture*. Isolating the confirmation process is a key insight that can simplify verifying other properties. The confirmation process is at the heart of many properties, and a small, easily understood confirmation process helps not just in verifying Property 2.

4.2 Detailed module descriptions

Voter authentication. After a voter signs in at a polling station, an election official would give that voter a voting token. In our implementation, we use a magnetic stripe card, but the token could also be a smartcard or a piece of paper with a printed security code. Each voting token is valid for only one voting machine. To begin voting, the voter inserts the token into the designated voting machine. The VoteCore module reads the contents of the token and verifies that the token is designated to work on this machine (via a serial number check), is intended for this particular election, has not been used with this machine before, and is signed using some public-key signature scheme. If the verification is successful, the VoteCore module communicates the contents of the voting token to the VoteSelection module.

Vote selection. The VoteSelection module parses the ballot definition file and interacts with the voter, allowing the voter to select candidates and vote on referenda. The voting token indicates which ballot to use, e.g., a Spanish ballot if the voter's native language is Spanish or a Democratic ballot if the voter is a Democrat voting in a primary. The VoteSelection module is intended to follow the rules outlined in the ballot definition file, e.g., allowing the voter to choose up to three candidates or to rank the candidates in order of preference. Of course, the VoteSelection module is untrusted and may contain malicious logic, so there is no guarantee that it operates as intended. The VoteSelection module interacts with the voter via the IOMultiplexor.

Vote confirmation. After the voter is comfortable with her votes, the VoteSelection module sends a description of the voter's preferences to the VoteConfirmation module. The VoteConfirmation module interacts with the voter via the IOMultiplexor, displaying a summary screen indicating the current selections and prompting the voter to approve or reject this ballot. If the voter approves, the VoteConfirmation module sends the ballot image[3] to the VoteCore module so it can be recorded. The VoteConfirmation module is constructed so that the data that the VoteConfirmation module sends to the VoteCore module is exactly the data that it received from the VoteSelection module.

Storing votes and canceling voter authentication tokens. After receiving a description of the votes from the VoteConfirmation module, the VoteCore atomically stores the votes and cancels the voter authentication token. Votes are stored on a durable, history-independent, tamper-evident, and subliminal-free vote storage mechanism [25]. By "atomically," we mean that once the VoteCore component begins storing the votes and canceling the authentication token, it will not be reset until after those actions complete. After those actions both complete, the VoteCore will trigger a reset by sending a

[3]A *ballot image* is merely a list of who this voter has voted for. It need not be an actual image or picture.

message to the ResetModule. Looking ahead, the only other occasion for the ResetModule to trigger a reset is when requested by VoteCore in response to a user wishing to cancel her voting session.

Cleaning up between sessions. Upon receiving a signal from the VoteCore, the ResetModule will reset all the other components. After those components awake from the reset, they will inform the ResetModule. After all components are awake, the ResetModule tells all the components to start, thereby initiating the next voting session and allowing the next voter to vote. We also allow the VoteCore module to trigger a reset via the ResetModule if the voter decides to cancel their voting process; when a voter triggers a reset in this way, the voter's authentication token is not canceled and the voter can use that token to vote again on that machine at a later time. Although the VoteCore has access to external media to store votes and canceled authentication tokens, all other state in this component is reset.

Enforcing a trusted path between the voter and the VoteConfirmation **module.** Although the above discussion only mentions the IOMultiplexor in passing, the IOMultiplexor plays a central role in the security of our design. Directly connecting the LCD and touch screen to both the VoteSelection module and the VoteConfirmation module would be unsafe: it would allow a malicious VoteSelection module to retain control of the LCD and touch screen forever and display a spoofed confirmation screen, fooling the voter into thinking she is interacting with the trusted VoteConfirmation module when she is actually interacting with malicious code. The IOMultiplexor mediates access to the LCD and touch screen to prevent such attacks. It enforces the invariant that only one module may have control over the LCD and touch screen at a time: either VoteConfirmation or VoteSelection may have control, but not both. Moreover, VoteConfirmation is given precedence: if it requests control, it is given exclusive access and VoteSelection is locked out. This allows our system to establish a trusted path between the voter interface and the VoteConfirmation module.

4.3 Hardware-enforced separation

Our architecture requires components to be protected from each other, so that a malicious VoteSelection component cannot tamper with or observe the state or code of other components. One possibility would be to use some form of software isolation, such as putting each component in a separate process (relying on the OS for isolation), in a separate virtual machine (relying on the VMM), or in a separate Java applet (relying on the JVM).

Instead, we use hardware isolation as a simple method for achieving strong isolation. We execute each module on its own microprocessor (with its own CPU, RAM, and I/O interfaces). This relies on physical isolation in an intuitive way: if two microprocessors are not connected by any communication channel, then they cannot directly affect each other. Verification of the interconnection topology of the components in our architecture consequently reduces to verifying the physical separation of the hardware and verifying the interconnects between them. Historically, the security community has focused primarily on software isolation because hardware isolation was viewed as prohibitively expensive [32]. However, we argue that the price of a microprocessor has fallen dramatically enough that today hardware isolation is easily affordable, and we believe the reduction in complexity easily justifies the extra cost.

With this approach to isolation, the communication elements between modules acquire special importance, because they determine the way that modules are able to interact. We carefully structured our design to simplify the connection topology as much as possible. Figure 3 summarizes the interconnectivity topology, and we describe several key aspects of our design below.

We remark that when multiple hardware components are used, one should ensure that the same versions of code run on each component.

Buses and wires. Our hardware-based architecture employs two types of communication channels: buses and wires. Buses provide high-speed unidirectional or bidirectional communication between multiple components. Wires are a simple signaling element with one bit of state; they can be either high or low, and typically are used to indicate the presence or absence of some event. Wires are unidirectional: one component (the sender) will set the value of a wire but never read it, and the other component (the receiver) will read the value of the wire but never set it. Wires are initially low, and can be set, but not cleared; once a wire goes high, it remains high until its controlling component is reset. We assume that wires are reliable but buses are potentially unreliable.

To deal with dropped or garbled messages without introducing too much complexity, we use an extremely simple communication protocol. Our protocol is connectionless and does not contain any in-band signaling (e.g., SYN or ACK packets). When a component in our architecture wishes to transmit a message, it will repeatedly send that message over the bus until it is reset or it receives an out-of-band signal to stop transmitting. The sender appends a hash of the message to the message. The receiver accepts the first message with a valid hash, and then acknowledges receipt with an out-of-band signal. This acknowledgment might be conveyed by changing a wire's value from low to high, and the sender can poll this wire to identify when to stop transmitting. Com-

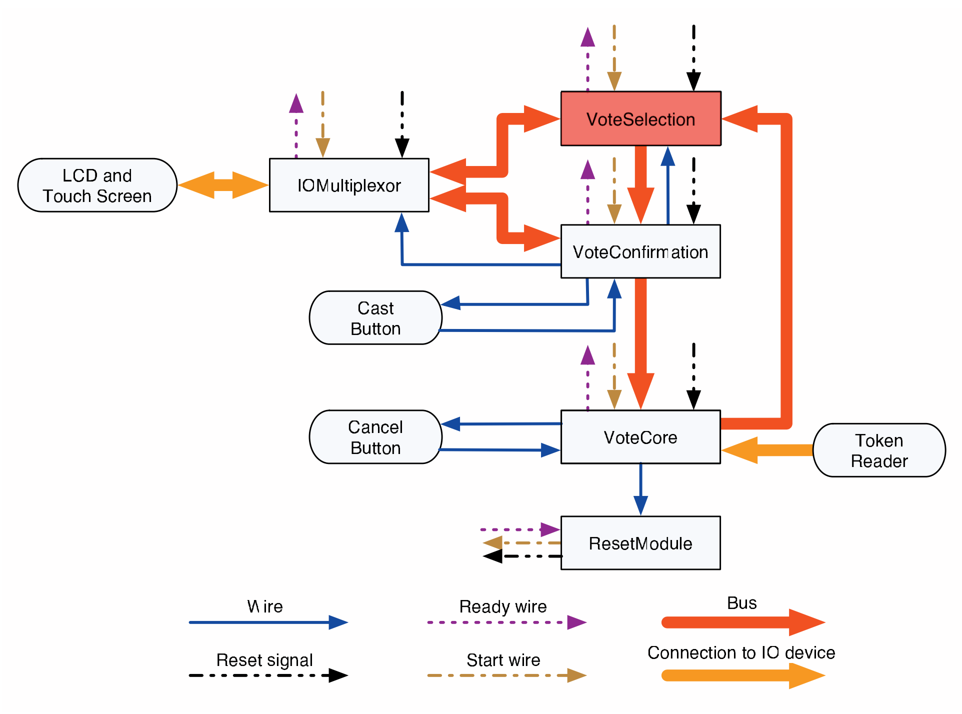


Figure 3: Our architecture, showing the hardware communication elements.

ponents that need replay protection can add a sequence number to their messages.

Using buses and wires. We now describe how to instantiate the communication paths in our high-level design from Section 4.2 with buses and wires. Once the VoteCore module reads a valid token, it repeatedly sends the data on the token to VoteSelection until it receives a message from VoteConfirmation. After storing the vote and canceling the authentication token, the VoteCore module triggers a reset by setting its wire to the ResetModule high.

To communicate with the voter, the VoteSelection component creates a bitmap of an image, packages that image into a message , and repeatedly sends that message to the IOMultiplexor. Since the VoteSelection module may send many images, it includes in each message a sequence number; this sequence number does not change if the image does not change. Also included in the message is a list of virtual buttons, each described by a globally unique button name and the x- and y-coordinates of the region. The IOMultiplexor will continuously read from its input source (initially the VoteSelection module) and draw to the LCD every bitmap that it receives with a new sequence number. The IOMultiplexor also interprets inputs from the touch screen, determines whether the inputs correspond to a virtual button and, if so, repeatedly writes the name of the region to the VoteSelection module until it has new voter input. Naming the regions prevents user input on one screen from being interpreted as input on a different screen.

When the voter chooses to proceed from the vote selection phase to the vote confirmation phase, the VoteConfirmation module will receive a ballot from the VoteSelection module. The VoteConfirmation module will then set its wire to the IOMultiplexor high. When the IOMultiplexor detects this wire going high, it will empty all its input and output bus buffers, reset its counter for messages from the VoteSelection module, and then only handle input and output for the VoteConfirmation module (ignoring any messages from VoteSelection). If the VoteConfirmation module determines that the user wishes to return to the VoteSelection module and edit her votes, the VoteConfirmation module will set its wire to the VoteSelection module high. The VoteSelection module will then use its bus to VoteConfirmation to repeatedly acknowledge that this wire is high. After receiving this acknowledgment, the VoteConfirmation module will reset itself, thereby clearing all internal state and also lowering its wires to the IOMultiplexor and VoteSelection modules. Upon detecting that this wire returns low, the IOMultiplexor will clear all its input and output buffers and return to handling the input and output for VoteSelection. The purpose for the handshake between the VoteConfirmation module and the VoteSelection module is to prevent the VoteConfirmation module from resetting and then immediately triggering on the receipt of the voter's previous selection (without this handshake, the VoteSelection module would continuously send the voter's previous selections, regardless of whether VoteConfirmation reset itself).

4.4 Reducing the complexity of trusted components

We now discuss further aspects of our design that facilitate the creation of implementations with minimal trusted code.

Resets. Each module (except for the ResetModule) interacts with the ResetModule via three wires, the initial values of which are all low: a ready wire controlled by the component and reset and start wires controlled by the ResetModule. The purpose of these three wires is to coordinate resets to avoid a situation where one component believes that it is handling the i-th voter while another component believes that it is handling the $(i+1)$-th voter.

The actual interaction between the wires is as follows. When a component first boots, it waits to complete any internal initialization steps and then sets the ready wire high. The component then blocks until its start wire goes high. After the ready wires for all components connected to the ResetModule go high, the ResetModule sets each component's start wire high, thereby allowing all components to proceed with handling the first voting session.

Upon completion of a voting session, i.e., after receiving a signal from the VoteCore component, the ResetModule sets each component's reset wire high. This step triggers each component to reset. The ResetModule keeps the reset wires high until all the component ready wires go low, meaning that the components have stopped executing. The ResetModule subsequently sets the reset wire low, allowing the components to reboot. The above process with the ready and start wires is then repeated.

Cast and cancel buttons. Our hardware architecture uses two physical buttons, a cast button and a cancel button. These buttons directly connect the user to an individual component, simplifying the task of establishing a trusted path for cast and cancel requests. Our use of a hardware button (rather than a user interface element displayed on the LCD) is intended to give voters a way to know that their vote will be cast. If we used a virtual cast button, a malicious VoteSelection module could draw a spoofed cast button on the LCD and swallow the user's vote, making the voter think that they have cast their vote when in fact nothing was recorded and leaving the voter with no way to detect this attack. In contrast, a physical cast button allows attentive voters to detect these attacks (an alternative might be to use a physical "vote recorded" light in the VoteCore). Additionally, if we used a virtual cast button, miscalibration of the touch screen could trigger accidental invocation of the virtual cast button against the voter's wishes. While calibration issues may still affect the ability of a user to scroll through a multi-screen confirmation process, we anticipate that such a problem will be easier to recover from than touch screen miscalibrations causing the DRE to incorrectly store a vote. To ensure that a malicious VoteSelection module does not trick the user into pressing the cast button prematurely, the VoteConfirmation module will only enable the cast button after it detects that the user paged through all the vote confirmation screens.

We want voters to be able to cancel the voting process at any time, regardless of whether they are interacting with the VoteSelection or VoteConfirmation modules. Since the VoteSelection module is untrusted, one possibility would be to have the IOMultiplexor implement a virtual cancel button or conditionally pass data to the VoteConfirmation module even when the VoteSelection module is active. Rather than introduce these complexities, we chose to have the VoteCore module handle cancellation via a physical cancel button. The cancel button is enabled (and physically lit by an internal light) until the VoteCore begins the process of storing a ballot and canceling an authentication token.

5 Prototype implementation

To evaluate the feasibility of the architecture presented in Section 4, we built a prototype implementation. Our prototype uses off-the-shelf "gumstix connex 400xm" computers. These computers measure 2cm by 8cm in size, cost $144 apiece, and contain an Intel XScale PXA255 processor with a 400 MHz StrongARM core, 64 MB of RAM, and 16 MB of flash for program storage. We enable hardware isolation by using a separate gumstix for each component in our architecture.

We do not claim that the gumstix would be the best way to engineer an actual voting system intended for use in the field. However, the gumstix have many advantages as a platform for prototyping the architecture. In conjunction with an equally sized expansion board, the processors support three external RS-232 serial ports, which transmit bidirectional data at 115200 kbps. We use serial ports as our buses. Additionally, each gumstix supports many general purpose input/output (GPIO) registers, which we use for our wires. Finally, the XScale

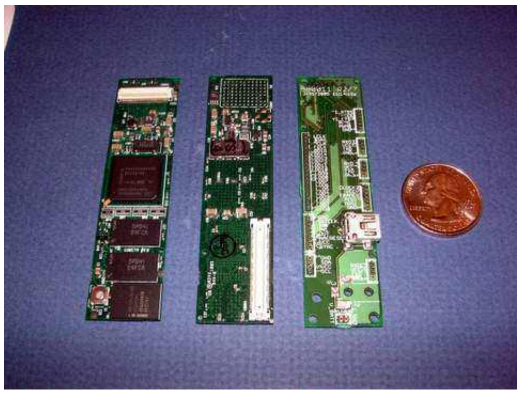


Figure 4: We show the front and back of a gumstix as well as an expansion board through which the GPIO and serial ports are soldered. The quarter gives an indication of the physical size of these components.

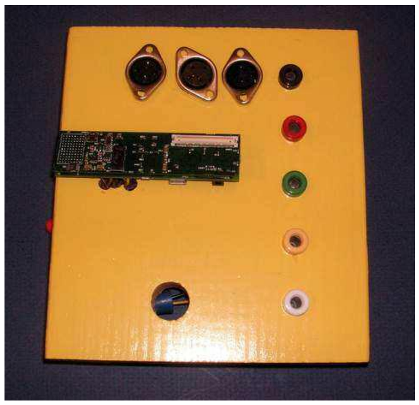


Figure 5: The mounting board for a single component. It contains three serial ports (along the top), 4 GPIO pins and a ground pin (along the right side), as well as a gumstix processor board mounted atop an expansion board.

processor supports an LCD and touch screen interface.

The gumstix platform's well-designed toolchain and software environment greatly simplified building our prototype. The gumstix, and our prototype, use a minimal Linux distribution as their operating system. Our components are written in Java and run on the Microdoc J9 Java VM; its JIT provides a significant speed advantage over the more portable JamVM Java interpreter. Our choice of Java is twofold: it is a type-safe language and so prevents a broad range of exploits; secondly, several program verification tools are available for verifying invariants in Java code [8, 19]. C# is another natural language choice since it too is type-safe and the Spec# [5] tool could aid in verification, but C# is not supported as well on Linux. We view a rich stable of effective verification tools to be just as important as type-safety in choosing the implementation language since software tools can improve confidence in the voting software's correctness. Both can eliminate large classes of bugs.

5.1 Implementation primitives

Our architecture requires implementations of two separate communications primitives: buses and wires. It is straightforward to implement buses using serial ports on the gumstix. To do so, we expose connectors for the serial ports via an expansion board connected to the main processor. Figures 4 and 5 show an example of such an expansion board. We additionally disable the `getty` terminal running on the serial ports to allow conflict free use of all three serial ports. The PXA255 processor has 84 GPIO pins, each controlled by registers; we implement wires using these GPIOs. A few of the pins are exposed on our expansion board and allow two components to be interconnected via their exposed GPIO pins. Each GPIO pin can be set in a number of modes. The processor can set the pin "high" so that the pin has a 3.3 volt difference between the reference ground; otherwise, it is low and has a 0 voltage difference between ground. Alternatively, a processor can poll the pin's state. To enforce the unidirectional communication property, particularly when a single wire is connected to more than two GPIOs, we could use a diode, which allows current to flow in only one direction [4]. We currently rely on software to enforce that once a GPIO is set high, it cannot ever be set low without first restarting the process; this is a property one could enforce in hardware via a latch, though our current prototype does not do so yet.

In addition to the GPIOs, the PXA255 exposes an NRESET pin. Applying a 3.3v signal to the NRESET pin causes the processor to immediately halt execution; when the signal is removed, the processor begins in a hard boot sequence. The gumstix are able to reboot in under 10 seconds without any optimizations, making the NRESET pin nearly ideal to clear a component's state during a reset. Unfortunately, the specifics of the reboot sequence causes slight problems for our usage. While the NRESET wire is held high, the GPIO pins are also high. In the case where one component reboots before another (or where selective components are reboot), setting the GPIOs high will inadvertently propagate a signal along the wire to the other components. Ideally, the pins would be low during reset. We surmise that designing a chip for our ideal reset behavior would not be difficult given

[4]Even this may not be enough, since an actual diode does not behave as the idealized diode we rely upon.

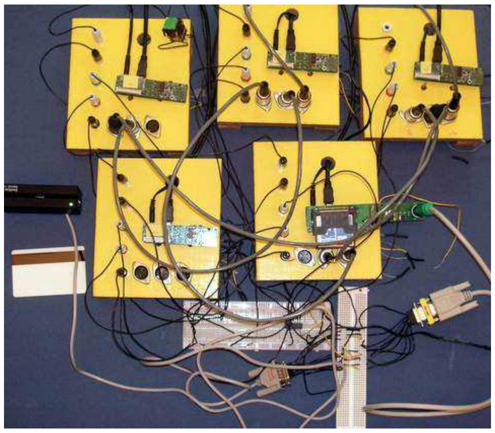
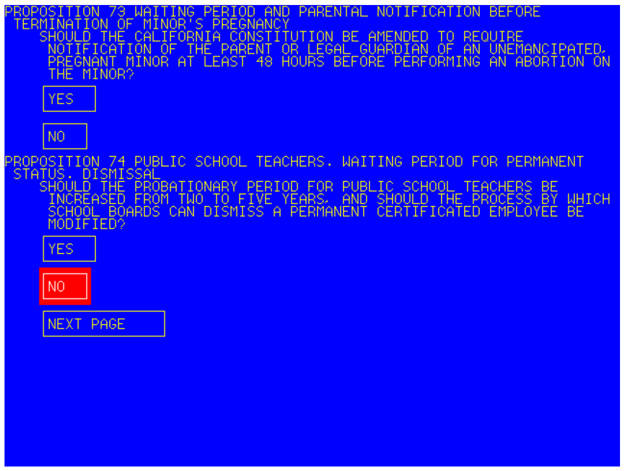


Figure 6: A picture of our prototype implementation. There is one board for each component in the system. The magnetic swipe card (along the left) is used for authentication, while the cast button is in the upper left component.



PROPOSITION 73 WAITING PERIOD AND PARENTAL NOTIFICATION BEFORE TERMINATION OF MINOR'S PREGNANCY

SHOULD THE CALIFORNIA CONSTITUTION BE AMENDED TO REQUIRE NOTIFICATION OF THE PARENT OR LEGAL GUARDIAN OF AN UNEMANCIPATED, PREGNANT MINOR AT LEAST 48 HOURS BEFORE PERFORMING AN ABORTION ON THE MINOR?

YES

NO

PROPOSITION 74 PUBLIC SCHOOL TEACHERS. WAITING PERIOD FOR PERMANENT STATUS. DISMISSAL

SHOULD THE PROBATIONARY PERIOD FOR PUBLIC SCHOOL TEACHERS BE INCREASED FROM TWO TO FIVE YEARS, AND SHOULD THE PROCESS BY WHICH SCHOOL BOARDS CAN DISMISS A PERMANENT CERTIFICATED EMPLOYEE BE MODIFIED?

YES [✓ NO]

Figure 7: The right image shows a screenshot of the VoteSelection component displaying referenda from the November 2005 election in Berkeley, CA. We flipped a coin to choose the response shown on this screen.

sufficient hardware expertise. Since the microprocessors in our platform do not exhibit our ideal behavior, in our prototype we have a separate daemon connected to an ordinary GPIO wire that stops the Java process running the component code when the reset pin goes high and then resets all wire state to low. The daemon starts a new component process when the signal to its reset pin is removed. This is just a way of emulating, in software, the NRESET semantics we prefer. Of course, a production-quality implementation would enforce these semantics in trusted hardware.

We use a Kanecal KaneSwipe GIT-100 magnetic card reader for authorizing voters to use the machine. A voter would receive a card with authentication information on it from poll workers upon signing in. The voter cannot forge the authentication information (since it contains a public key signature), but can use it to vote once on a designated DRE. The reader has an RS-232 interface, so we are able to use it in conjunction with the serial port on the gumstix.

Finally, our implementation of the VoteCore component uses a compact flash card to store cast ballot images and invalid magcard identifiers. Election officials can remove the flash card and transport it to county headquarters after the close of polls. A deployed DRE might use stronger privacy-protection mechanisms, such as a history-independent, tamper-evident, and subliminal-free data structure [25]. For redundancy, we expect a deployed DRE to also store multiple copies of the votes on several storage devices. A full implementation of the VoteSelection component would likely also use some kind of removable storage device to store the ballot definition file. In our prototype, we hard-code a sample ballot definition file into the VoteSelection component. This suffices for our purposes in gauging the feasibility of other techniques.

Our prototype consists of five component boards wired together in accordance with Figure 3. We implement all of the functionality except for the cancel button. See Figure 6 for a picture showing the five components and all of their interconnections. Communication uses physical buses and wires. The I/O multiplexer, after each update operation, sends an image over a virtual bus connected (connected via the USB network) to the PC for I/O. It sends the compressed image it would ordinarily blit to the framebuffer to the PC so that the PC can blit it to its display. The gumstix only recently supported LCD displays, and we view our PC display as an interim solution. The additional software complexity for using the LCD is minimal as it only requires blitting an image to memory.

Figure 7 shows our voting software running on the gumstix. We used ballot data from the November 2005 election in Alameda County, California.

6 Evaluation

6.1 Verifying the desired properties

Property 1. Recall that to achieve "memorylessness" we must be able to show the DRE is always reset after a voter has finished using the machine, and the DRE only opens a given file read-only or write-only, but not

```
1   grabio.set();
2   ...  UPDATE DISPLAY ...
3   castenable.set();
4   if (cast.isSet()) {
5      while (true) {
6         toVoteCore.write(ballot);
7      }
8   }
```

Confirm.java

```
1   byte [] ballot =
2          fromVoteConf.read();
3   if (ballot != null) {
4      ...  INVALIDATE VOTER TOKEN ...
5      ballotbox.write (ballot);
6      while (true) {
7          resetWire.set();
8      }
9   }
```

VoteCore.java

Figure 8: Code extracts from the VoteConfirmation and VoteCore modules, respectively. Examining these code snippets with the connection topology helps us gain assurance that the architecture achieves Properties 1 and 2.

both. To show that the DRE is reset after storing a vote, we examine a snippet of the source code from `VoteCore.java`, the source code for the VoteCore module in Figure 8. In line 7, after storing the ballot into the ballot box, the VoteCore module continuously raises the reset wire high. Looking at the connection diagram from Figure 3, we note the reset wire terminates at the ResetModule and induces it to restart all components in the system. Further inspecting code not reproduced in Figure 8 reveals the only reference to the `ballotbox` is in the constructor and in line 5, so writes to it are confined to line 5.

Finally, we need merely examine every file open call to make sure they are either read-only or write only. In practice, we can guarantee this by ensuring writable files are append-only, or for more sophisticated vote storage mechanisms as proposed by Molnar et al., that the storage layer presents a write-only interface to the rest of the DRE.

Property 2. For the "consent-to-cast" property, we need to verify two things: 1) the ballot can only enter the VoteCore through the VoteConfirmation module, and 2) the voter's consent is required before the ballot can leave the VoteConfirmation module.

Looking first at `Confirm.java` in Figure 8, the VoteConfirmation module first ensures it has control of the touch screen as it signals the IOMultiplexor with the "grabio" wire. It then displays the ballot over the bus, and subsequently enables the cast button. Examining the hardware will show the only way the wire can be enabled is through a specific GPIO, in fact the one controlled by the "castenable" wire. No other component in the system can enable the cast button, since it is not connected to any other module. Similarly, no other component in the system can send a ballot to the VoteCore module: on line 6 of `Confirm.java`, the VoteConfirmation sends the ballot on a bus named "toVoteCore", which is called the "fromVoteConf" bus in `VoteCore.java`. The ballot is demarshalled on line 1. Physically examining the hardware configuration confirms these connections, and shows the ballot data structure can only come from the VoteConfirmation module. Finally, in the VoteCore module, we see the only use of the ballotbox is at line 5 where the ballot is written to the box. There are only two references to the BallotBox in the `VoteCore.java` source file (full file not shown here), one at the constructor site and the one shown here. Thus we can be confident that the only way for a ballot to be passed to the BallotBox is if a voter presses the cast button, indicating their consent. We must also verify that the images displayed to the voter reflect the contents of the ballot.

	Java	C (JNI)	Total
Communications	2314	677	2991
Display	416	52	468
Misc. (interfaces)	25	0	25
VoteSelection	377	0	377
VoteConfirmation	126	0	126
IOMultiplexor	77	0	77
VoteCore	846	54	900
ResetModule	121	0	121
Total	4302	783	5085

Table 1: Non-comment, non-whitespace lines of code.

6.2 Line counts

One of our main metrics of success is the size of the trusted computing base in our implementation. Our code contains shared libraries (for communications, display, or interfaces) as well as each of the main four modules in the TCB (VoteConfirmation, IOMultiplexor, VoteCore, and ResetModule). The VoteSelection module can be excluded from the TCB when considering Properties 1 and 2. Also included in the TCB, but not our line count figures, are standard libraries, operating system code, and JVM code.

In Table 1, we show the size of each trusted portion as a count of the number of source lines of code, excluding comments and whitespace.

The communications libraries marshal and unmarshal data structures and abstract the serial devices and GPIO pins. The display libraries render text into our user interface (used by the VoteConfirmation component) and ultimately to the framebuffer.

7 Applications to VVPATs and cryptographic voting protocols

So far we've been considering our architecture in the context of a stand-alone paperless DRE machine. However, jurisdictions such as California require DREs to be augmented with a voter verified paper audit trail. In a VVPAT system, the voter is given a chance to inspect the paper audit trail and approve or reject the printed VVPAT record. The paper record, which remains behind glass to prevent tampering, is stored for later recounts or audits.

VVPAT-enabled DREs greatly improve integrity protection for non-visually impaired voters. However, a VVPAT does not solve all problems. Visually impaired voters who use the audio interface have no way to visually verify the selections printed on the paper record, and thus receive little benefit from a VVPAT. Also, a VVPAT is only an integrity mechanism and does not help with vote privacy. A paper audit trail cannot prevent a malicious DRE from leaking one voter's choices to the next voter, to a poll worker, or to some other conspirator. Third, VVPAT systems require careful procedural controls over the chain of custody of paper ballots. Finally, a VVPAT is a fall-back, and even in machines that provide a VVPAT, one still would prefer the software to be as trustworthy as possible.

For these reasons, we view VVPAT as addressing some, but not all problems. Our methods can be used to ameliorate some of the remaining limitations, by providing better integrity protection for visually impaired voters, better privacy protection for all voters, reducing the reliance on procedures for handling paper, and reducing the costs of auditing the source code. Combining our methods with a VVPAT would be straightforward: the VoteConfirmation module could be augmented with support for a printer, and could print the voter's selections at the same time as they are displayed on the confirmation screen. While our architecture might be most relevant to jurisdictions that have decided, for whatever reason, to use paperless DREs, we expect that our methods could offer some benefits to VVPAT-enabled DREs, too.

Others have proposed cryptographic voting protocols to enhance the security of DREs [10, 16, 26, 27]. We note that our methods could be easily combined with those cryptographic schemes.

8 Extensions and discussion

In addition to the properties we discussed, there are other relevant security properties which we considered in designing our voting system. We have not rigorously validated that the design provides these properties, though we outline directions we will follow to do so.

Property 3 *The DRE cannot leak information through the on-disk format. Additionally, it should be history-independent and tamper evident.*

Property 3 removes the back-end tabulation system from the trusted path. Without this property, the tabulation system may be in the trusted path because the data input to the tabulation system may reveal individual voter's choices. With this property, it is possible to make the outputs of each individual DRE publicly available, and allow multiple parties to independently tabulate the final results. We believe we can use the techniques from Molnar et al. in implementing Property 3 [25].

Property 4 *The DRE only stores ballots the voter approves.*

Property 4 refers to a few conditions. The DRE must not change the ballot after the voter makes their selection in the VoteSelection module; software analysis techniques could prove useful in ensuring the ballot is not modified. Additionally, there will need to be some auditing of the code to ensure display routines accurately display votes to the screen.

Property 5 *The ballot contains nothing more than the voter's choices.*

In particular, the ballot needs to be put into a canonical form before being stored. Violation of Property 5 could violate the voter's privacy, even if the voter approves the ballot. Suppose the voter's choice, "James Polk" were stored with an extra space: "James␣␣Polk". The voter would not likely notice anything were amiss, but this could convey privacy leaking information in a subliminal channel [16]. We expect software analysis techniques could ensure that canonicalization functions are run on all program paths. Combined with Property 4 to ensure the ballot doesn't change, this would help ensure the ballot is canonicalized.

We do not expect these to be an exhaustive list of the desirable security properties; rather, they are properties that we believe are important and that we can easily achieve with this architecture without any changes.

Minimizing the underlying software platform. Our prototype runs under an embedded Linux distribution that is custom designed for the gumstix platform. Despite its relatively minimal size (4MB binary for kernel and all programs and data), it still presents a large TCB, most of which is unnecessary for a special-purpose voting appliance. We expect that a serious deployment would dispense with the OS and build a single-purpose embedded application running directly on the hardware. For instance, we would not need virtual memory, memory protection, process scheduling, filesystems, dual-mode operation, or most of the other features found in general-purpose operating systems. It might suffice to have a simple bootloader and a thin device driver layer specialized to just those devices that will be used during an election. Alternatively, it may be possible to use ideas from nanokernels [11], microkernels [14, 31], and operating system specialization [30] to reduce the operating system and accordingly the TCB size.

Deploying code. Even after guaranteeing the software is free of vulnerabilities, we must also guarantee that the image running on the components is the correct image. This is not an easy problem, but the research community has begun to address the challenges. SWATT [34] is designed to validate the code image on embedded platforms, though their model does not allow for CPUs with virtual memory, for example. TCG and NGSCB use a secure hardware co-processor to achieve the same ends, though deploying signed and untampered code to devices still requires much work. Additionally, a human must then check that all components are running the latest binary and must ensure that the binaries are compatible with each other – so that a version 1.0 VoteCore is not running with a version 1.1 IOMultiplexor module, for example.

This concern is orthogonal to ours, as even current voting machines must deal with versioning. It illustrates one more challenge in deploying a secure voting system.

9 Related work

There has been a great deal of work on high-assurance and safety-critical systems, which are designed, implemented, and tested to achieve specific safety, reliability, and security properties. We use many classic techniques from that field, including minimization of the size of the TCB and decomposing the application into clearly specified components. One contribution of this paper is that we show in detail how those classic techniques may be applied in the e-voting context.

Modularity is widely understood to be helpful in building high-reliability systems. Deep space applications often use multiple components for reliability and fault tolerance [41]. Telephone switches use redundant components to upgrade software without loss of availability [41]. In avionics, Northrop Grumman has proposed an architecture for future avionics systems suitable to the Department of Defense's Joint Vision 2020 [39]. Their MLS-PCA architecture is intended to support tens to hundreds of thousands of separate processors. MLS-PCA uses isolation for several purposes, including mission flexibility, multi-level security when interoperating with NGOs, and reduction in the amount of trusted software over traditional federated architectures. Of these reasons, the last is most related to our setting. Others have articulated composibility of security as one of the key challenges in applying modularity to the security setting [21].

Rebooting is widely recognized in industry as a useful way to prevent and rectify errors [9]. Rebooting returns the system to its original state, which is often a more reliable one. Others use preventative rebooting to mitigate resource leaks and *Heisenbugs* [15]. In contrast, our work uses rebooting for what we believe is a new purpose: privacy. Prior work focuses on availability and recoverability, while we use it to simplify our task in verifying privacy preserving properties.

The Starlight Interactive Link is a hardware device that allows a workstation trusted with secret data to safely interact with an unclassified network [2]. The Starlight Interactive Link acts as a data diode. A chief concern is secret data leaking onto the untrusted network. Many of these ideas led to the design of our IOMultiplexor.

Our design shares similarities with existing DRE voting machines from major vendors, such as Diebold, Hart Intercivic, Sequoia Voting Systems, and Election Systems and Software. A criticism of the machines, however, is that people must trust the software running on the machines since the voter cannot be sure their vote was properly recorded. Rebecca Mercuri has called for vendors to augment DRE machines with a voter verified paper audit trail (VVPAT) [22, 23]. In this DRE variant, the voter must approve a paper copy of their selections that serves as the permanent record. The paper copy is typically held behind glass so the voter cannot tamper with it. Even in spite of malicious software, the paper copy accurately reflects the voter's selections.

The principle of isolation for systems is well established [4, 7, 12, 20, 28, 31, 32, 35, 36, 40]. Isolation has been proposed as a technique to improve security in two existing voting systems. The FROGS and Pnyx.DRE systems both separate the vote selection process from vote confirmation [1, 6]. However, FROGS significantly alters the voting experience while it is not clear the Pnyx.DRE was designed for verification nor does it provide our privacy protections.

Finally, Hall discusses the impact of disclosing the source for voting machines for independent audit [13].

10 Conclusions

Democracy deserves the best efforts that computer scientists can deliver in producing accurate and verifiable voting systems. In this work, we have proposed better DRE based voting designs, whether VVPAT-enabled or not. In both cases, our architecture provides stronger security properties than current voting systems.

Our approach uses hardware to isolate components from each other and uses reboots to guarantee voter privacy. In particular, we have shown how isolating the VoteSelection module, where much of the hairiness of a voting system resides, into its own module can eliminate a great deal of complex code from the TCB. Though isolation is not a novel idea, the way we use it to improve the security of DREs is new. This work shows that it is possible to improve existing DREs without modifying the existing voter experience or burdening the voter with additional checks or procedures.

The principles and techniques outlined here show that there is a better way to design voting systems.

Acknowledgments

We thank David Jefferson for early discussions that led to this work. Umesh Shankar, Chris Karlof, Rob Johnson, and Mike Reiter provided invaluable feedback on the presentation, organization, and provided great insight into the thinking and organization of the work. We thank Matt Bishop and David Molnar for helpful discussions about related works.

References

[1] Auditability and voter-verifiability for electronic voting terminals. `http://www.scytl.com/docs/pub/a/PNYX.DRE-WP.pdf`, December 2004. White paper.

[2] M. Anderson, C. North, J. Griffin, R. Milner, J. Yesberg, and K. Yiu. Starlight: Interactive Link. In *Proceedings of the 12th Annual Computer Security Applications Conference (ACSAC)*, 1996.

[3] J. Bannet, D. W. Price, A. Rudys, J. Singer, and D. S. Wallach. Hack-a-vote: Demonstrating security issues with electronic voting systems. *IEEE Security and Privacy Magazine*, 2(1):32–37, Jan./Feb. 2004.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Sstems Principles (SOSP 2003)*, October 2003.

[5] M. Barnett, K. R. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proceedings of Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS)*, 2004.

[6] S. Bruck, D. Jefferson, and R. Rivest. A modular voting architecture ("Frogs"). `http://www.vote.caltech.edu/media/documents/wps/vtp_wp3.pdf`, August 2001. Voting Technology Project Working Paper.

[7] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, October 1997.

[8] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. R. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[9] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – a technique for cheap recovery. In *6th Symposium on Operating System Design and Implementation (OSDI)*, December 2004.

[10] D. Chaum. Secret-ballot receipts: True voter-verifiable elections. *IEEE Security & Privacy Magazine*, 2(1):38–47, Jan.–Feb. 2004.

[11] D. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, October 1995.

[12] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 6th USENIX Security Symposium*, August 1996.

[13] J. Hall. Transparency and access to source code in e-voting. Unpublished manuscript.

[14] G. Heiser. Secure embedded systems need microkernels. *USENIX ;login*, 30(6):9–13, December 2005.

[15] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing*, 1995.

[16] C. Karlof, N. Sastry, and D. Wagner. Cryptographic voting protocols: A systems perspective. In *Fourteenth USENIX Security Symposium*, August 2005.

[17] T. Kohno, A. Stubblefield, A. D. Rubin, and D. S. Wallach. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy*, pages 27–40, May 2004.

[18] M. Kuhn. Optical time-domain eavesdropping risks of CRT displays. In *IEEE Symposium on Security and Privacy*, May 2002.

[19] G. Leavens and Y. Cheon. Design by contract with JML. `ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf`.

[20] J. Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70, September 1996.

[21] D. McCullough. Noninterference and the composability of security properties. In *IEEE Symposium on Security and Privacy*, May 1988.

[22] R. Mercuri. *Electronic Vote Tabulation Checks & Balances*. PhD thesis, School of Engineering and Applied Science of the University of Pennsylvania, 2000.

[23] R. Mercuri. A better ballot box? *IEEE Spectrum*, 39(10):46–50, October 2002.

[24] D. Mertz. XML Matters: Practical XML data design and manipulation for voting systems. `http://www-128.ibm.com/developerworks/xml/library/x-matters36.html`, June 2004.

[25] D. Molnar, T. Kohno, N. Sastry, and D. Wagner. Tamper-evident, history-independent, subliminal-free data structures on PROM storage -or- How to store ballots on a voting machine (extended abstract). In *IEEE Symposium on Security and Privacy*, May 2006.

[26] C. A. Neff. A verifiable secret shuffle and its application to e-voting. In *8th ACM Conference on Computer and Communications Security (CCS 2001)*, pages 116–125, November 2001.

[27] C. A. Neff. Practical high certainty intent verification for encrypted votes. `http://www.votehere.net/vhti/documentation`, October 2004.

[28] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[29] RABA Innovative Solution Cell. Trusted agent report Diebold AccuVote-TS voting system, January 2004.

[30] M. Rajagopalan, S. Debray, M. Hiltunen, and R. Schlichting. Automated operating system specialization via binary rewriting. Technical Report TR05-03, University of Arizona, February 2005.

[31] R. Rashid Jr., A. Tevanian, M. Young, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1987.

[32] J. Rushby. Design and verification of secure systems. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP)*, December 1981.

[33] Science Applications International Corporation (SAIC). Risk assessment report Diebold AccuVote-TS voting system and processes, September 2003.

[34] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWAtt: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[35] M. Swift, M. Annamalai, B. Bershad, and H. Levy. Recovering device drivers. In *Proceedings of the 6th ACM/USENIX Symposium on Operating System Design and Implementation*, December 2004.

[36] M. Swift, B. Bershad, and H. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Sstems Principles (SOSP 2003)*, October 2003.

[37] W. van Eck. Electromagnetic radiation from video display units: An eavesdropping risk? *Computers & Security*, 4, 1985.

[38] D. Wagner, D. Jefferson, M. Bishop, C. Karlof, and N. Sastry. Security analysis of the Diebold AccuBasic interpreter. California Secretary of State's Voting Systems Technology Assessment Advisory Board (VSTAAB), February 2006.

[39] C. Weissman. MLS-PCA: A high assurance security architecture for future avionics. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003)*, 2003.

[40] A. Whitaker, M. Shaw, and S. Gribble. Denali: A scalable isolation kernel. In *10th ACM SIGOPS European Workship*, September 2002.

[41] I.-L. Yen and R. Paul. Key applications for high-assurance systems. *IEEE Computer*, 31(4):35–45, April 1998.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering technical excellence and innovation
- encouraging computing outreach in the community at large
- providing a neutral forum for the discussion of critical issues

Membership Benefits

- Free subscription to *;login:,* the Association's magazine, both in print and online
- Online access to all Conference Proceedings from 1993 to the present
- Access to the USENIX Jobs Board: Perfect for those who are looking for work or are looking to hire from the talented pool of USENIX members
- The right to vote in USENIX Association elections
- Discounts on technical sessions registration fees for all USENIX-sponsored and co-sponsored events
- Discounts on purchasing printed Proceedings, CD-ROMs, and other Association publications
- Discounts on industry-related publications: see http://www.usenix.org/membership/specialdisc.html

For more information about membership, conferences, or publications, see http://www.usenix.org.

SAGE, a USENIX Special Interest Group

SAGE is a Special Interest Group of the USENIX Association. Its goal is to serve the system administration community by:

- Establishing standards of professional excellence and recognizing those who attain them
- Promoting activities that advance the state of the art or the community
- Providing tools, information, and services to assist system administrators and their organizations
- Offering conferences and training to enhance the technical and managerial capabilities of members of the profession

Find out more about SAGE at http://www.sage.org.

Thanks to USENIX & SAGE Supporting Members

Addison-Wesley Professional/ Prentice Hall Professional
Ajava Systems, Inc.
AMD
Cambridge Computer Services, Inc.
EAGLE Software, Inc.
Electronic Frontier Foundation
Eli Research
FOTO SEARCH Stock Footage and Stock Photography
GroundWork Open Source Solutions
Hewlett-Packard
IBM
Infosys
Intel
Interhack
Microsoft Research
MSB Associates
NetApp
Oracle
OSDL
Raytheon
Ripe NCC
Sendmail, Inc.
Splunk
Sun Microsystems, Inc.
Taos
Tellme Networks
UUNET Technologies, Inc.

ISBN 1-931971-45-5